FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Plataforma Industrial de Internet das Coisas

Eliseu Moura Pereira

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: Prof. Gil Manuel Magalhães de Andrade Gonçalves

Coorientador: Rui Pedro Ferreira Pinto

Coorientador: João Pedro Correia dos Reis

24 de Julho de 2018

Resumo

As tecnologias de informação e comunicação são consideradas uma das áreas mais importantes nas últimas décadas e a previsão é que assim continue em anos futuros. A constante miniaturização da tecnologia possibilitou o desenvolvimento de dispositivos de processamento de informação, onde computadores de grandes dimensões foram substituídos por computadores pessoais de pequenas dimensões, desde PCs, tablets, telemóveis e outros sistemas embebidos, que podem ser parte integrante de produtos de maior dimensão. Estes sistemas embebidos apresentam várias características, desde a recolha de informação, a partir de sensores, módulos de processamento de informação e interfaces de comunicação, inclusive conetividade à Internet, ambos com e sem fios.

No contexto industrial, atualmente tem-se vindo a alocar vários esforços no desenvolvimento de soluções que potenciam a 4ª revolução industrial, mais conhecida por Indústria 4.0. Esta iniciativa assenta sobre os conceitos de *Internet of Things* (IoT) aplicados a ambientes fabris. Neste caso, equipamentos e dispositivos físicos no chão de fábrica são representados virtualmente por entidades lógicas, mais conhecidas por *Digital Twin*, que têm por objetivo normalizar comunicações entre equipamentos – *Machine to Machine Communication* (M2M), de forma a tomar decisões descentralizadas de alto nível o mais autonomamente possível. Desta forma, estes equipamentos são capazes de fazer convergir o mundo físico com o mundo virtual, apresentando capacidades e funcionalidades enaltecidas pela sua virtualização e interação descentralizada em rede. Estes sistemas são mais conhecidos como sistemas de produção ciber-físicos – *Cyber-Physical Production Systems* (CPPS).

Uma das maiores dificuldades atualmente refere-se ao M2M, nomeadamente na forma como equipamentos são capazes de descobrir outros equipamentos, bem como plataformas que fornecem um conjunto de serviços. A comunicação e partilha de informação entre dispositivos de chão de fábrica em tempo real apresenta alguns desafios, nomeadamente a interoperabilidade de diferentes dispositivos e protocolos de comunicação, gestão descentralizada de dispositivos e serviços conectados à Internet, evitando pré-configurações da rede.

De forma a resolver estes problemas, pretende-se com esta dissertação introduzir aos equipamentos de chão de fábrica capacidades de *plug and play*, isto é, equipamentos que, sempre que se ligam à rede, são capazes de automaticamente descobrir e estabelecer comunicação, de forma a consumir e fornecer serviços, com outros dispositivos presentes na rede, bem como dispositivos presentes noutras redes. Para isso, foi desenvolvida uma plataforma capaz de efetuar a gestão de dispositivos presentes em diferentes redes, designada de *Cyber-Physical Production System Sniffer* (*CPPS Sniffer*).

O *CPPS Sniffer* consiste numa aplicação distribuída desenvolvida em Java, que utiliza o protocolo MQTT, que é capaz de gerir vários dispositivos presentes numa rede privada, de forma a potenciar a comunicação M2M local. Para além disso, considerando diferentes redes, em que cada uma delas possui um *CPPS Sniffer*, é possível haver comunicação M2M entre diferentes dispositivos presentes em diferentes redes, usando a Internet e os respetivos *CPPS Sniffer* das redes para comunicar. Desta forma, é possível criar e gerir diferentes redes e os seus dispositivos de uma

forma descentralizada, evitando aumentar a complexidade dos próprios dispositivos.

No final, através dos testes efetuados, concluiu-se que a plataforma, *CPPS Sniffer*, cumpre os requisitos estipulados tanto a nível de uma rede local, como remotamente ou seja com múltiplas redes locais. Quando é sujeita a comparação com outras plataformas IoT, fica saliente que se trata de uma componente mais distribuída que as restantes, ou seja funciona ao nível de *Fog Computing*.

Abstract

Information and communication technologies are considered one of the most important areas of the last decades and it's predicted to continue to be in future years. The constant miniaturization of the technology has made possible the development of data processing devices, where computers of large dimensions were replaced by personal computers of small dimensions, such as PCs, tables, smartphones, and other embedded systems, that can be integrated in bigger products. These embedded systems have lots of technical features, such as capture of data from sensors, data processing modules and communication interfaces, and also Internet connection, wireless or wired cable.

In an industrial context, nowadays people are increasing efforts on developing solutions for the 4th industrial revolution, known as Industry 4.0. This initiative is based on the concepts of the Internet of Things (IoT) applied to manufacturing environments. In that case, equipments and physical devices in the low level of the factory are represented virtually by logical entities, known as Digital Twin, that have the goal to normalize communication between equipments – Machine to Machine Communication (M2M), in order to make high level decentralized decisions. This way, these equipments are able to connect the physical to the virtual word, by showing skills and functionalities to virtualize and decentralize the network. These systems are known as Cyber-Physical Production Systems (CPPS).

One of the biggest challenges nowadays is the M2M communication, particularly in the way that equipments are capable to find out other equipments and also other services platforms. The communication and share of data, in real time, between devices has some challenges, mainly in the continuous device communication, the use of different protocols, the decentralized management of devices and services through the Internet.

To solve those problems, this dissertation will introduce to the devices the skills of plug and play. Every time they connect to the network, they are capable to automatically discovery and establish a connection with other devices inside or outside the local network, by consuming and provide services. For that, it was developed a platform capable of doing the management of devices in multiple networks, named Cyber-Physical Production System Sniffer (CPPS Sniffer).

The CPPS Sniffer is a distributed application, that was developed in Java, which uses the MQTT protocol, that is capable to manage several devices, in order to potencialize the M2M communication. Furthermore, regarding different networks, in which every single one of them has a CPPS Sniffer, it's possible to have M2M communication between different devices in different networks, by using the Internet and the CPPS Sniffers. This way, is possible create and manage different networks and their devices in a distributed way, avoiding the increase of device complexity.

In conclusion, the platform CPPS Sniffer, meets the requirements, both in a single local network and in a multiple remote network. When compared with other IoT platforms, CPPS Sniffer is more distributed, in another words it works at fog computing level.

Agradecimentos

O desenvolvimento desta dissertação foi um processo muito enriquecedor a nível académico, pois graças a ele desenvolvi inúmeras capacidades. Durante todo este período, recebi constante apoio e disponibilidade por parte do meu orientador o Professor Gil Gonçalves, e dos meus coorientadores Rui Pinto e João Reis, tanto dando-me conselhos e opiniões, como esclarecendo-me dúvidas.

Gostava também de agradecer o apoio, que recebi nestes 5 anos, por parte da minha família e dos meus amigos, que estiveram comigo, tanto nos bons, como nos maus momentos.

Eliseu Moura Pereira

"However difficult life may seem, there is always something you can do and succeed at.
It matters that you don't just give up."

Stephen Hawking

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

IoT	Internet of Things
IIoT	Industrial Internet of Things
CPS	Cyber-Physical Systems
CPPS	Cyber-Physical Production Systems
M2M	Machine to Machine
ERP	Enterprise Resource Planning
MES	Manufacturing Execution Systems
HTTP	Hypertext Transfer Protocol
REST	Representational state transfer
MQTT	Message Queue Telemetry Transport
QoS	Quality of Service
CoAP	Constrained Application Protocol
AMQP	Advanced Message Queuing Protocol
OPC-UA	Open Platform Communications - Unified Architecture
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
AWS	Amazon Web Services
GEs	Generic Enablers
IDS	Industrial Data Space
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
LAN	Local Area Network
JSON	JavaScript Object Notation
XML	Extensible Markup Language
SQL	Structured Query Language
NoSQL	Not Only Structured Query Language

Capítulo 1

Introdução

Atualmente estamos a assistir à 4ª revolução industrial intitulada de Indústria 4.0, que consiste na otimização de todos os processos dentro de uma fábrica, por forma a aumentar tanto a produção como a qualidade dos produtos. A Indústria 4.0 baseia-se em conceitos como a *Internet of Things* (IoT), que é utilizado para conectar todos os dispositivos, como o sistema ciber-físico (CPS) que permite monitorizar/virtualizar um processo físico, ou como a *cloud* que constitui um fornecedor de serviços ou centro de armazenamento de dados.

1.1 Motivação

Os processos de produção de uma fábrica, estão condicionados a limites temporais, limitando estes a leitura por parte dos sensores, a atuação de componentes e a comunicação entre estes. Tendo em conta que os sensores e os atuadores evoluíram para se tornarem cada vez mais rápidos seja na leitura das entradas ou na escrita das saídas, as comunicações tendem a tornar-se mais eficientes também tanto a nível de atrasos como a nível de *overhead* de pacotes na rede.

Neste momento, muitas das redes IoT utilizam protocolos orientados para tecnologias Web, nas quais são irrelevantes tanto os atrasos de rede como o congestionamento desta mesma, pois apesar disso, estes protocolos permitem a partilha de serviços entre as entidades envolvidas. Assim o principal objetivo da Indústria 4.0 é otimizar todo o processo de fabrico além de permitir ao utilizador acesso a tudo o que está a acontecer na fábrica, sejam braços robóticos, sensores ou outro tipo de componente envolvida no processo de fabrico. Sendo que para isso são necessários protocolos leves, robustos e confiáveis.

Além disso, o número de dispositivos tem vindo a aumentar nas fábricas, seja para monitorizar o estado das máquinas, a qualidade dos produtos ou a quantidade de matéria prima. Com isto, fica mais difícil fazer a gestão destas redes, pois aumentou o número de dispositivos e a sua complexidade. Quando se fala em gestão de uma rede, isto consiste em garantir que todos os dispositivos presentes nesta estão em condições de comunicar entre si.

1.2 Contexto

Atualmente, os padrões de procura por produtos em diferentes mercados, caracterizam-se pela exigência de elevados níveis de customização e personalização dos produtos, sendo que o ciclo de vida destes é tendencialmente cada vez mais curto. Do ponto de vista das empresas e fabricantes destes produtos, a solução passa por estratégias orientadas para a produção em massa, o que implica requisitos de processos industriais críticos, como resiliência, robustez e reconfiguração automática. Uma dessas estratégias está relacionada com a implementação de sistemas de produção ciber-físicos – *Cyber-Physical Production Systems* (CPPS), o que permite reconfigurações dinâmicas de equipamentos de chão de fábrica, bem como controlo descentralizado do chão de fábrica. O CPPS é um sistema composto por elementos computacionais que permite a gerir e controlar de componentes físicas. Quando aplicado à industria este sistema permite a virtualização de toda a fábrica desta forma disponibiliza mecanismos para monitorizar e controlar remotamente todas as operações. Este tipo de sistemas são capazes de responder rapidamente a variações não esperadas da procura, com base em mecanismos de tomada de decisões autónoma e descentralizada.

Avanços na implementação e desenvolvimento de CPPS levou recentemente a alavancar o conceito de Indústria 4.0, que marca a 4ª revolução industrial, que se caracteriza pela fusão de várias tecnologias, como por exemplo IoT, *Cloud Computing*, *Service Oriented Architecture* (SOA) e *Big Data*. Um dos principais objetivos da Indústria 4.0 consiste em conectar todos os objetos físicos do chão de fábrica entre si, sendo que cada um deve ser capaz de se anunciar aos restantes objetos e iniciar comunicações com estes, de forma a trocar informação. Para isso, o sistema físico é adaptado e transformado num CPS, onde representações virtuais – *Digital Twin* são usadas para anunciar as capacidades físicas de cada objeto, identificar e comunicar com outros objetos, dentro de uma rede descentralizada de *Digital Twins*, mais conhecida como *Industrial Internet of Things* (IIoT).

Transformar equipamentos e dispositivos em CPS traz vantagens na descoberta de outros dispositivos na rede, mas também na forma como esses dispositivos podem ser integrados em plataformas externas, como *Manufacturing Execution Systems* (MES), *Big Data*, *Enterprise Resource Planning* (ERP) e outras plataformas *cloud*. Com recurso a arquiteturas SOA, CPS recolhem informação física relevante sobre o equipamento e/ou processo, baseada em vários sensores, sendo que essa informação é enviada posteriormente para estas plataformas externas, de forma a ser processada. Muitas plataformas *cloud* consomem esta informação, ao fornecer serviços específicos sobre a informação gerada a baixo nível. Estes serviços podem alimentar mecanismos de apoio à decisão, que enviam de volta para o CPS comandos, com ações especificas a serem executadas em diferentes atuadores, que têm um impacto concreto sobre os processos físicos que estavam a ser monitorizados em primeiro lugar.

Neste tipo de sistemas, é normal existir uma importante fase de especificações e desenho da arquitetura antes do desenvolvimento, de forma a identificar quais os protocolos de comunicação mais vantajosos a utilizar em diferentes cenários, bem como o recurso à implementação de plataformas IoT, de forma a gerir de uma forma mais abstrata o CPPS. As funcionalidades principais

que se pretendem são a gestão de *Digital Twins*, a descoberta de serviços e funcionalidades e o uso de um protocolo de comunicação que facilite a troca de informação entre equipamentos – M2M e plataformas *cloud*, de uma forma segura e robusta.

1.3 Objetivos

O principal objetivo desta dissertação é, num contexto industrial, a implementação de uma aplicação, designada de *CPPS Sniffer*, capaz de gerir, de uma forma *plug and play*, diferentes dispositivos, equipamentos industriais e plataformas *cloud*, possibilitando a interação e comunicação entre diferentes entidades presentes na rede, sem a necessidade de haver configurações manuais da rede, como especificação de IPs, arquiteturas de rede, etc. A grande vantagem do *CPPS Sniffer* é a capacidade de gerir dispositivos presentes em diferentes redes, ligadas entre si via Internet. Desta forma, inicialmente elaborou-se um estudo sobre os principais protocolos de comunicação e plataformas IoT utilizados em aplicações IoT, mais concretamente a nível industrial, de forma a perceber quais as principais funcionalidades dos protocolos e das plataformas IoT já estariam disponíveis nas soluções existentes e de que forma se poderia tirar proveito dessas soluções existentes para implementar uma aplicação que respondesse a vários requisitos industriais. A solução desenvolvida deve cumprir vários requisitos, nomeadamente:

- **Descoberta de serviços** - O conceito de *plug and play* consiste na configuração automática de dispositivos e outras entidades sempre que se ligam a uma rede, de forma a reconhecer e ser reconhecido por outros dispositivos lá presentes. Sendo que um dos objetivos é facilitar a comunicação de dados entre diferentes entidades fabris, pretende-se que a descoberta de serviços vá para além da descoberta de serviços local a uma rede interna, fornecendo as mesmas capacidades quando diferentes entidades estão ligadas a diferentes redes, via Internet.

- **Interoperabilidade dos protocolos** - Considerando que diferentes dispositivos utilizam diferentes protocolos de comunicação, do ponto de vista da adaptabilidade e escalabilidade da solução implementada, esta deve suportar diferentes protocolos de comunicação, de forma a normalizar as interações entre dispositivos heterogéneos.

- **Gestão de *Digital Twins*** - Sabendo que um CPPS e/ou IIoT consiste num conjunto de diversas representações virtuais de objetos físicos, que estão disponíveis em diferentes redes, é essencial que a solução desenvolvida tenha um mecanismo de gestão de *Digital Twins*, de forma a facilitar a identificação e interação entre estes.

- **Interfaces de comunicação entre dispositivos e plataformas** - M2M e entre equipamentos e plataformas externas, como *cloud*, MES e ERP, facilitando a partilha de informação, do ponto de vista das entidades que publicam ou subscrevem a informação de uma forma completamente distribuída.

1.4 Definição do problema

Como foi referido anteriormente, as redes onde se conectam diferentes dispositivos de chão de fábrica e plataformas têm vindo a ficar mais complexas. Um exemplo de uma arquitetura genérica é representada na Figura - 1.1. Nesta arquitetura, estão representados uma variedade de sensores, nomeadamente pressão, humidade e temperatura, distribuídos por duas redes locais distintas. Para além disso, essas redes locais estão conectadas, via Internet, a duas plataformas *clouds* externas, um manipulador robótico acessível remotamente e uma interface de interação com o utilizador final, de forma a monitorizar e controlar remotamente diferentes processos.

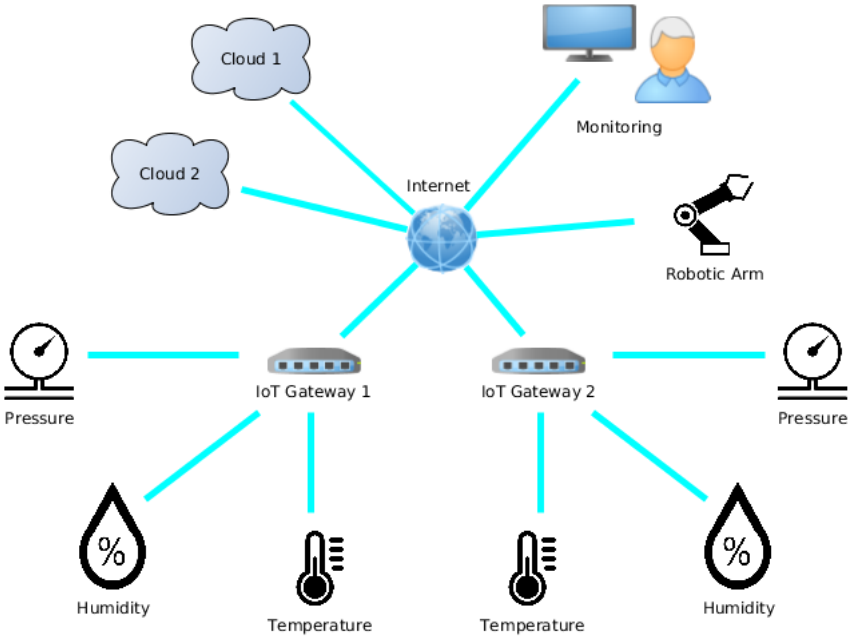

Figura 1.1: Arquitetura física do sistema IoT.

Do ponto de vista de interação e partilha de informação entre todas estas entidades, um desafio que se coloca é a própria comunicação entre as entidades. Caso não hajam mecanismos de *plug and play*, será necessário configurar antecipadamente a arquitetura da rede, nomeadamente configurar manualmente, em todas as entidades, os seus endereços IP fixos e abrir portas de comunicação, bem como guardar configurações sobre os endereços IP e portas de todos as entidades para as quais se pretenda que haja interação. Mecanismos *plug and play* resolvem este problema, ao fazer a gestão dos parâmetros associados às entidades na rede, e permitem que estas sejam transparentes em toda a rede. Contudo, mecanismos *plug and play* existentes em protocolos de comunicação e plataformas IoT só estão disponíveis na situação em que a comunicação existe entre entidades presentes na mesma rede local, o que consiste numa limitação em aplicações CPPS e IIoT, onde é frequente existirem várias redes heterogéneas.

O segundo maior problema centra-se na dificuldade de fazer a gestão de várias entidades na rede de uma forma descentralizada. Normalmente os protocolos de comunicação e plataformas

IoT mais utilizados funcionam numa filosofia centralizada, ou parcialmente centralizada, o que torna o processo de reestruturação da rede em tempo real, em caso de falhas de componentes, uma tarefa difícil, se não impossível de realizar. A gestão da rede deve ser feita de uma forma completamente descentralizada, de forma a haver transparência na forma como a arquitetura da rede pode ser reestruturada, perante falhas parciais de rede.

1.5 Hipótese

Baseado na revisão de literatura, identificam-se as funcionalidades de *plug and play* e descoberta de serviços, como base para uma boa gestão de CPPS, no âmbito de fábricas inteligentes e sistemas avançados de produção. Contudo, mecanismos atuais que fornecem este tipo de funcionalidades, presentes em certos protocolos de comunicação de plataformas IoT, apresentam limitações no âmbito da gestão apenas em redes locais. A hipótese formulada é que sistemas de manufatura avançada, baseados em CPPS e IIoT, que envolvem por natureza várias redes heterogéneas, é possível implementar um mecanismo, que tire partido das funcionalidades de *plug and play* presentes em protocolos de comunicação atuais, e que estenda estas funcionalidades para suportar a descoberta de serviços remotamente, isto é, considerando dispositivos presentes em redes diferentes, ligadas via Internet.

1.6 Estrutura do documento

O presente documento está estruturado em 5 capítulos diferentes. O Capítulo 1 refere-se à introdução, onde é feita uma curta apresentação do tema da dissertação, uma descrição da motivação que levou ao desenvolvimento desta dissertação, o contexto do problema, os objetivos pretendidos, uma breve definição do problema e a descrição da hipótese.

No Capítulo 2 faz-se o levantamento do estado da arte, ao identificar e descrever os principais protocolos de comunicação usados a nível industrial, assim como as plataformas IoT usadas para gestão de dispositivos e Digital Twins, finalizando o capítulo com uma comparação ao nível dos protocolos e dos *middlewares* identificados. No que toca a protocolos IoT são descritos vários protocolos que adotam modelos cliente-servidor e publicador-subscritor, como o *Message Queue Telemetry Transport* (MQTT), o *Constrained Application Protocol* (CoAP), o *Advanced Message Queuing Protocol* (AMQP) e o *Open Platform Communications - Unified Architecture* (OPC-UA). Quanto às plataformas IoT foram estudadas o FIWARE, o *Amazon Web Services* (AWS) IoT, o *Microsoft Azure*, entre outros.

O Capítulo 3 foca-se na descrição dos trabalhos referentes ao desenvolvimento da aplicação *CPPS Sniffer*. O Capítulo 4 refere-se aos testes e resultados, neste capitulo são documentados os resultados obtidos em diferentes arquiteturas, havendo lugar para comparação e discussão dos resultados obtidos com os estudados durante a revisão bibliográfica, tanto a nível de parâmetros

como a nível de arquitetura. Finalmente, o Capítulo 5 refere-se às conclusões do trabalho realizado, bem como a identificação e descrição de melhorias que se podem efetuar na solução final, em formato de trabalho futuro.

Capítulo 2

Revisão Bibliográfica

A importância da virtualização dos dispositivos e/ou serviços num CPS é crucial para que exista uma visão transparente de toda a rede e para a fácil integração de plataformas externas. A aplicação de um CPS num ambiente industrial, facilita imenso a constante reestruturação das componentes presentes na fábrica, pois permite a configuração automática de novos dispositivos.

Assim para uma correta virtualização de um CPS é necessário estabelecer quais as tecnologias a utilizar, bem como as vantagens e desvantagens de cada uma. Para isso, foi efetuado o levantamento do estado da arte, que é dividido em três tópicos distintos, nomeadamente os principais protocolos usados em aplicações IIoT, que plataformas IoT estão disponíveis para gerir CPPS e propostas de arquiteturas industriais, que incluem capacidades de gestão e descoberta de dispositivos e serviços remotamente. Dentro de cada tópico, inicialmente vão ser descritos os protocolos (MQTT, CoAP, AMQP e OPC-UA), as plataformas IoT (*Microsoft Azure*, AWS IoT, FIWARE e outras plataformas IoT) e as arquiteturas (IDS). Posteriormente, à descrição serão analisadas as vantagens e desvantagens tanto dos protocolos, como das plataformas IoT, como das arquiteturas.

2.1 Protocolos IoT

Como foi referido anteriormente, numa aplicação IIoT é crucial a especificação do protocolo a utilizar nas comunicações e interações entre dispositivos e plataformas, seja pela grande quantidade de dados enviados, ou pela necessidade de curtos tempos de reposta por parte dos dispositivos. Assim, na presente secção são apresentados os protocolos de comunicação principais existentes em aplicações IIoT e CPPS.

2.1.1 MQTT

O protocolo MQTT foi desenvolvido pela IBM, de forma a ser utilizado em sistemas IoT. Neste momento existem duas versões deste protocolo, o MQTT e o MQTT-SN, sendo o último orientado a redes de sensores.

No que diz respeito ao MQTT [18] é utilizado num modelo *publish/subscribe* composto por publicadores, *brokers* e subscritores. Neste caso, publicadores são entidades que partilham informação, enquanto os subscritores são as entidades que recebem a informação partilhada. A forma de funcionamento é simples, os publicadores enviam as suas mensagens para um *broker*, atualizando dados referentes a um dado tópico, que por sua vez trata de difundir as mensagens por todos os subscritores, notificando-os sempre que existe uma nova atualização no tópico subscrito [31]. A Figura - 2.1 representa os vários campos de uma mensagem do protocolo MQTT.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td colspan="4">Tipo de Mensagem</td><td>DUP</td><td colspan="2">QoS</td><td>Retain</td></tr>
<tr><td colspan="8">Tamanho Restante da Mensagem</td></tr>
<tr><td colspan="8">Header variável (opcional)</td></tr>
<tr><td colspan="8">Payload (opcional)</td></tr>
</table>

Figura 2.1: Formato de mensagem MQTT.

Na mensagem do protocolo MQTT, o *header* ocupa os primeiros 2 bytes da mensagem. Os primeiros 4 bits são ocupados com o tipo de mensagem que vai ser enviada podendo esta variar consoante o pedido que seja feito. O bit seguinte é ocupado por uma *flag* (DUP), que é ativada caso seja necessário retransmitir a mensagem, esta ativação apenas acontece caso o valor da qualidade de serviço seja maior que 0. O valor da qualidade de serviço preenche os bits 5 e 6, sendo que este parâmetro indica a fiabilidade da entrega da mensagem ao seu destinatário. O último bit deste primeiro byte contém a *flag* de retenção que é utilizado nas mensagens de PUBLISH, para o servidor que recebeu a mensagem esperar que esta seja recebida pelos atuais subscritores, antes de a apagar. O resto da mensagem é ocupada com o *header* opcional e o *payload*. O *header* opcional é preenchido consoante o tipo de mensagem que é enviado, mas por norma tem um identificador da mensagem e parâmetros semelhantes. Quanto ao *payload*, este contém os dados que vão ser enviados.

Como foi referido anteriormente, para cada tipo de mensagem existe uma funcionalidade associada. Neste caso, os diferentes tipos de mensagens existentes são:

- **CONNECT**: O cliente solicita uma conexão ao servidor, para isso ele coloca o tipo de mensagem a 1 e como *payload* da mensagem introduz um identificador do cliente, o seu nome de utilizador e a sua palavra-passe, para isso é necessário ativar os bits correspondentes a cada campo do *header* opcional. O servidor responde com uma mensagem de CONACK.

- **CONNACK**: Mensagem retornada pelo servidor ao cliente indicando o sucesso ou a negação da conexão.

- **PUBLISH**: As mensagens de PUBLISH estão sempre associadas a um tópico, elas são envidas de um cliente para o servidor, que depois difunde o seu conteúdo pelos outros clientes que subscreveram esse tópico. Esta mensagem contém como *header* variável o tópico da mensagem bem como um identificador desta. Como *payload* contém os dados que pretende

publicar. A resposta a esta mensagem pode ser um PUBACK, caso a variável QoS esteja a 1, neste caso a mensagem pode ser difundida pelos subscritores, ou um PUBREC, caso a variável QoS esteja a 2, neste particular os dados da mensagem ainda não podem ser partilhados com os outros subscritores.

- **PUBACK**: Esta mensagem consiste no envio de um *acknowledge* por parte do servidor para o cliente ou de um subscritor para o servidor. Quando é recebida esta mensagem o seu recetor pode eliminar a mensagem de PUBLISH enviada anteriormente.
- **PUBREC**: Este tipo de mensagem tem como objetivo assegurar-se de que o cliente pretende fazer a publicação. Quando esta mensagem é recebida o recetor envia uma mensagem PUBREC com o mesmo ID da mensagem de PUBLISH.
- **PUBREL**: Quando é recebida esta mensagem o servidor fica autorizado para efetuar a publicação, ele responde ao cliente com uma mensagem de PUBCOMP.
- **PUBCOMP**: Quando esta mensagem é recebida a anterior mensagem de PUBLISH pode ser eliminada pois já foi difundida.
- **SUBSCRIBE**: Para subscrever um ou vários tópicos, um cliente envia um pedido de SUBSCRIBE para um servidor. O conteúdo desta mensagem é composto pelo *header* característico desta mensagem, pelo *header* variável onde é indicado o identificador da mensagem e pelo *payload* onde se encontra o tópico que se pretende subscrever com o respetivo nível de QoS pretendido. Quando o servidor recebe esta mensagem ele responde com um SUBACK.
- **SUBACK**: A mensagem SUBACK corresponde a um *acknowledge* enviado pelo servidor para o cliente. Esta mensagem contém a QoS garantida para cada um dos tópicos ordenada pela mesma ordem em que os tópicos estavam na mensagem de SUBSCRIBE.
- **UNSUBSCRIBE**: Esta mensagem é enviada pelo cliente para o servidor e contém os tópicos que pretender deixar de subscrever. O servidor responde a esta mensagem com um UNSUBACK.
- **UNSUBACK**: Confirmação por parte do servidor que os tópicos já não estão subscritos.
- **PINGREQ**: Verificação se o dispositivo ainda está operacional.
- **PINGRESP**: Resposta ao PINGREQ.
- **DISCONNECT**: Permite desconectar o cliente do servidor.

A Figura - 2.2 representa-se um diagrama de sequência da troca das diferentes mensagens identificadas, em relação ao protocolo MQTT.

O MQTT tornou-se num dos protocolos mais utilizado em redes IoT, seja para a comunicação M2M entre os dispositivos, ou para a comunicação entre os dispositivos e as plataformas externas, sendo também adotado para aplicações industriais. Peralta et al. [23] projetaram um sistema

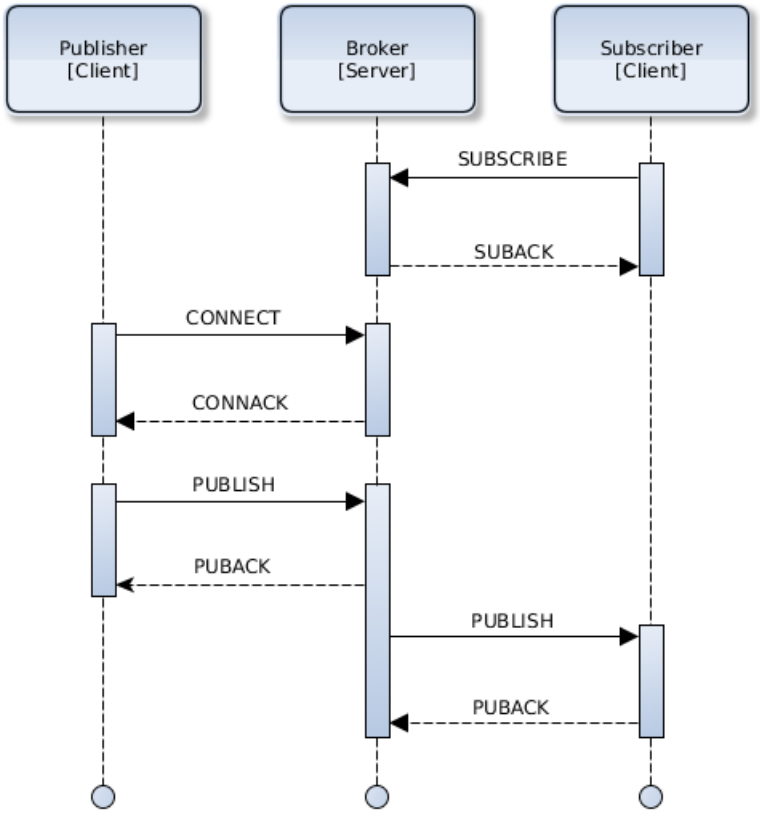

Figura 2.2: Diagrama de sequência da troca de mensagens no protocolo MQTT.

industrial composto por dispositivos IoT, IoT *gateways* e *cloud*, em que o protocolo que efetuava a intermediação era o MQTT. O principal objetivo desta implementação era baixar os consumos energéticos da rede, adotando uma arquitetura descentralizada (*Fog Computing*), utilizando para isso IoT *gateways* que efetuavam uma parte do processamento, de maneira a não sobrecarregar a *cloud*. Um outro trabalho foi elaborado por Katsikeas et al. [12], onde é comparado o MQTT a outros protocolos IoT, e no final é implementado um sistema IoT, que utiliza um modelo *publish/-subscribe*, sendo este aplicado ao controlo e à monitorização de turbinas eólicas.

2.1.1.1 MQTT-SN

Quanto ao MQTT-SN, utiliza as mesmas componentes referidas anteriormente (2.1.1), estando esta variante do protocolo responsável por fazer a comunicação entre os clientes e as *gateways*. Contudo, tanto as comunicação entre as *gateways* e o *broker* como entre o *broker* e servidor, baseiam-se na versão normal do MQTT.

2.1.2 CoAP

No que diz respeito ao CoAP [29], este protocolo foi desenvolvido para ser suportado por dispositivos com menor capacidade de processamento, podendo estes mesmos serem microcon-

troladores 8 bits, caracterizados por possuir pouco poder computacional, nomeadamente a nível de capacidades de ROM e RAM. Relativamente às suas funcionalidades, uma das principais é a partilha de serviços, além da fácil integração com o protocolo HTTP. Este protocolo é idêntico ao REST no que toca a partilha de serviços, apesar de para tal utilizar UDP, devido aos baixos *overheads* verificados no uso de UDP em comparação com o TCP.

A utilização de UDP na camada de transporte, tem como principal consequência a falta de fiabilidade na entrega das mensagens. Dessa forma, houve a necessidade de implementar um mecanismo de confirmação das mensagens, que obriga o recetor da mensagem a reenvia-la para o emissor, em que a mensagem reenviada tem o mesmo identificador que a recebida anteriormente. Desta maneira, fica saliente que o protocolo CoAP utiliza um modelo *request/reply*, a não ser que, o servidor não consiga processar o pedido imediatamente, desta forma é enviado um simples *acknowledge*, e quando processar finalmente o pedido, envia a confirmação. Esta característica é importante quando se lida com redes sobrecarregadas. Para não existir confusão da parte do servidor sobre a que pedido corresponde certa resposta, visto este receber inúmeros pedidos, cada par de mensagens pedido/resposta tem um *token* associado, tendo este o mesmo valor nas 2 mensagens. O formato de uma mensagem CoAP está subdividida em diferentes partes, como representado na Figura - 2.3.

<table>
<tr><td>0-1</td><td>2-3</td><td>4-5</td><td>6-7</td><td>8-9</td><td>10-11</td><td>12-13</td><td>14-15</td></tr>
<tr><td>Versão</td><td>Tipo</td><td colspan="2">Tamanho Token</td><td colspan="4">Código</td></tr>
<tr><td colspan="8">ID Mensagem</td></tr>
<tr><td colspan="8">Token (opcional)</td></tr>
<tr><td colspan="8">Opções (opcional)</td></tr>
<tr><td colspan="8">Payload (opcional)</td></tr>
</table>

Figura 2.3: Formato de mensagem CoAP.

Quanto ao *header* fixo, este é constituído por dois primeiros bits, que indicam a versão do protocolo CoAP que está a ser utilizada, os dois bits seguintes indicam de que tipo é a mensagem, sendo que o tipo de mensagem pode ser confirmável, não confirmável, *acknowledgement* ou reset e de seguida quatro bits que indicam o tamanho do *token*. Para além disso, são reservados oito bits para o código, e por fim um identificador da mensagem que ocupa dezasseis bits. De destacar que o código presente contém informação útil para caracterizar a mensagem, ou seja se é um pedido, se ocorreu algum erro, se é uma resposta.

Este protocolo mantém imensas semelhanças com o protocolo HTTP, o que pode ser visto como uma grande vantagem, pois muitos dos serviços Web atuais são baseados nesse protocolo, e assim uma possível substituição pelo CoAP seria muito mais facilitada. Assim os métodos utilizados por este protocolo para troca de mensagens são idênticos aos do HTTP. Neste caso, os métodos identificados são:

- **GET**: Este método envia um pedido, para que lhe seja retornada a informação correspondente ao recurso nele indicado.

- **PUT**: Neste método é enviado um pedido para criar ou atualizar um determinado recurso.
- **POST**: O método POST é semelhante ao PUT, diferindo apenas no facto em que no caso do POST, a informação enviada é processada pelo recetor antes de ser disponibilizada.
- **DELETE**: O método DELETE consiste num pedido de remoção do recurso especificado no pedido.

Quanto a mecanismos de descoberta de serviços, o protocolo CoAP implementa um sistema baseado em *multicast* para encontrar na rede o dispositivo que pretende. Desta forma, é enviado um pedido, com um recurso associado, para um endereço *multicast* onde todos os dispositivos estão ligados. De seguida, os dispositivos que tiverem informação sobre esse recurso, iniciam um canal de comunicação com o dispositivo que enviou o pedido.

Hu [10] implementou um sistema de monitorização, constituído por uma rede wireless de sensores industriais, que estão ligados a *gateways*, que por sua vez enviam a informação recolhida pelos sensores para uma *cloud*. O sistema está divido entre uma componente responsável pelo controlo e outra pela gestão dos dados. Relativamente aos resultados, esta arquitetura IIoT foi comparada a uma semelhante que utiliza o AWS IoT, sendo que no que se refere a latência, a arquitetura que utiliza CoAP, apresenta uma redução de 40% quando comparado com o AWS IoT.

Por sua vez, Konieczek et al. [13] testaram a aplicação deste protocolo a um sistema de sincronização de relógios, sendo que estes sistemas são caracterizados por elevadas restrições temporais, tal como acontece nas aplicações industriais. Este sistema é modelado por uma arquitetura *request/reply*, em que teria de ser garantido o mesmo valor do relógio em todos os clientes. Em termos de resultados conseguiram obter uma diferença máxima de 2 milissegundos, o que prova serem resultados bastante bons.

Tanto o CoAP como o MQTT foram implementados para redes com larguras de banda mais limitadas, e dispositivos com poucos recursos. Na implementação realizada por Thangavel et al. [30], onde se comparam estes dois protocolos, ficou explicito a nível de resultados, que para redes com elevado número de perdas de pacotes, o mais aconselhável seria a utilização de MQTT, pois é mais fiável, enquanto que, para redes com baixo nível de perdas de pacotes, o mais adequado é o CoAP.

2.1.3 AMQP

O AMQP [21] é um protocolo IoT direcionado para a implementação de comunicações entre diferentes componentes presentes em sistemas empresariais distribuídos. Apesar de ter sido desenvolvido com uma aplicação diferente, este protocolo adequa-se na perfeição a sistemas IoT, pois é mais leve do que o HTTP em relação a *overheads*, com a vantagem de ser fiável na entrega das mensagens, pois o AMQP utiliza TCP/IP na camada de transporte.

Na Figura - 2.4, está representado a arquitetura utilizada na implementação deste protocolo, que é constituída tanto por publicadores, que são dispositivos que atualizam e enviam constantemente o seu estado para um servidor, como por subscritores, que são de igual forma dispositivos

que subscrevem certos tópicos que lhes vão sendo disponibilizados pelo servidor. Quanto ao servidor, ou AMQP *broker*, este é constituído por duas componentes fundamentais:

- ***Exchange***: Esta componente é responsável tanto por aceitar as mensagens provenientes dos publicadores, como de posteriormente reencaminhar essas mesmas mensagens para as filas correspondentes aos subscritores, que subscreveram os tópicos associados a estas mensagens.

- ***Message Queue***: As filas de mensagens presentes na Figura - 2.4, têm como função guardar as mensagens antes das entregar aos subscritores. Estas filas são implementadas de diferentes formas, por exemplo como uma fila privada de subscrições, ou seja, vai guardando mensagens de múltiplos publicadores, para posteriormente as entregar a um único subscritor.

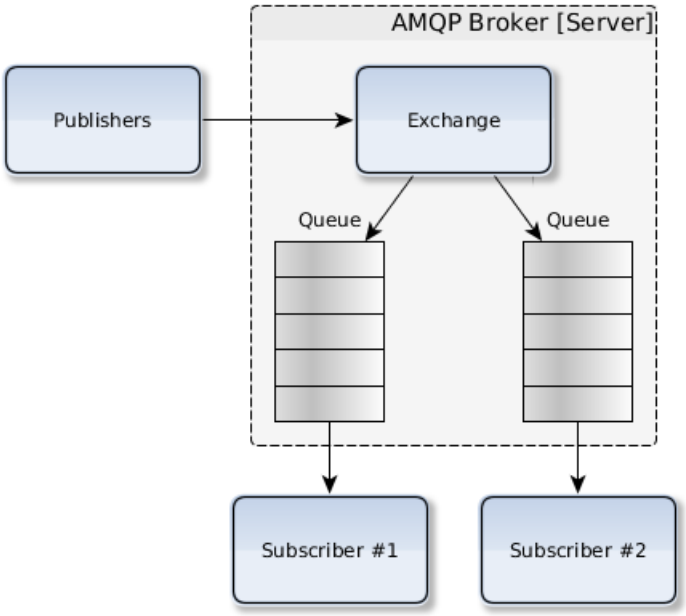

Figura 2.4: Arquitetura utilizada pelo protocolo AMQP.

O formato de uma mensagem AMQP é representado na Figura - 2.5, onde está presente um *header* fixo de 7 bytes, constituído por 4 campos o tipo, o canal, o tamanho da mensagem (*payload*) e o *frame-end* que permite verificar que a mensagem não contem erros.

0	1	2	3	4	5	6
Tipo	Canal		Tamanho			
Payload (n bytes)						
Frame-end (1 byte)						

Figura 2.5: Formato de mensagem AMQP em bytes.

Relativamente aos tipos de mensagens, existem quatro tipos possíveis, as de METHOD, as de HEADER, as de BODY e as de HEARTBEAT, desta forma o conteúdo do *payload* vai variar consoante o tipo de mensagem.

- **METHOD**: Esta mensagem consiste no pedido de execução de um método, para isso é necessário enviar o ID tanto da classe como do método, bem como os argumentos para execução desse método.

- **HEADER/BODY**: Este conjunto de duas mensagens permite partilhar uma estrutura de dados com os atributos pretendidos, a estrutura destes dados vem definida na mensagem de HEADER, enquanto que os dados estão presentes nas mensagens de BODY consequentes.

- **HEARTBEAT**: As mensagens de HEARTHBEAT servem como o nome indica para o servidor verificar de uma maneira rápida quais os dispositivos estão desconectados.

2.1.4 OPC-UA

O OPC-UA é um protocolo que tem como principal mercado a área das aplicação distribuídas industriais. Este protocolo é o mais utilizado por aplicações industriais, pois foi projetado com o propósito de satisfazer as necessidades destas mesmas aplicações. Este protocolo utiliza um modelo *request/reply*, onde as transações são despoletadas pelo cliente, ao qual o servidor responde com o valor pretendido (caso seja uma operação de leitura), ou com uma mensagem de sucesso (caso seja uma operação de escrita). Salienta-se que é o servidor que faz a comunicação com o hardware, seja para ler entradas ou escrever saídas. Sendo da família OPC este protocolo evoluiu, na medida em que melhorou certos aspetos, nomeadamente:

- **Independência de plataforma**: As anteriores versões do OPC corriam obrigatoriamente numa plataforma Microsoft, o que condicionava o uso desta tecnologia. Com o surgimento do OPC-UA, conseguiu-se que tanto os clientes como os servidores pudessem correr em qualquer plataforma, independente da arquitetura.

- **Sistemas embebidos**: Ainda relacionado com a independência de plataforma, também é possível correr um servidor OPC-UA numa plataforma embebida, com poucos recursos computacionais.

- **Aumento de segurança**: Quanto à segurança foi implementado um sistema de autorização/autenticação ao nível do canal cliente/servidor, onde este processo utiliza chaves de segurança públicas.

- **Melhora na modelação de dispositivos**: Relativamente à modelação de dispositivos, o OPC-UA expandiu o número de atributos utilizados para modelar dispositivos, como também permite exprimir relações entre componentes.

A descoberta de serviços neste tipo de arquitetura funciona através de um servidor local para descoberta de serviços. É neste servidor que todos os outros registam os serviços que suportam, assim como representado na Figura - 2.6.

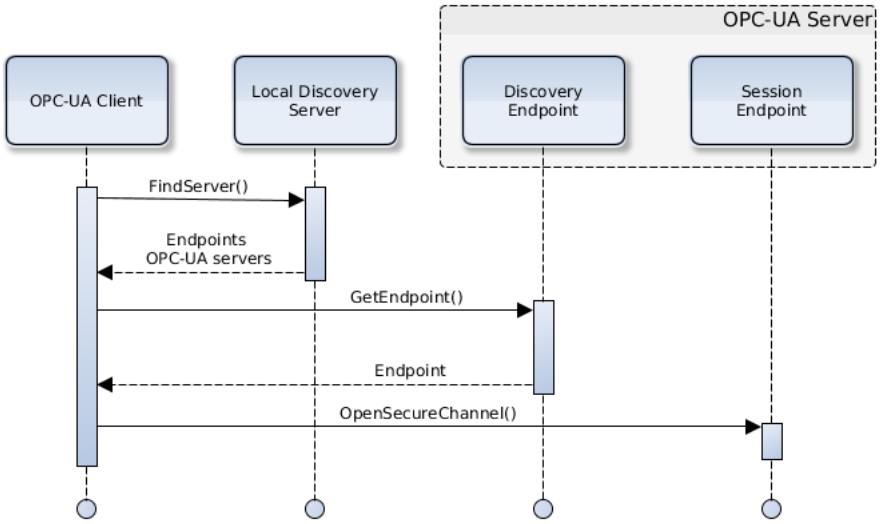

Figura 2.6: Diagrama de sequência da descoberta de serviços no OPC-UA

Quando o servidor local para descoberta de serviços recebe um pedido de um cliente, verifica quais os servidores na rede que suportam esse serviço e retorna ao cliente os detalhes para se ligar a esse mesmo servidor. Quanto ao servidor, este contem 2 componentes, uma responsável por retornar aos clientes os detalhes necessários para iniciar uma comunicação e uma outra pela qual são enviados os comandos (leitura/escrita).

O OPC-UA suporta 2 formatos de mensagem: um o formato binário e outro o formato XML. O formato define como os dados são codificados antes de serem enviados, assim tanto o emissor como o recetor devem ser capazes de codificar e descodificar a mensagem. Sendo que na camada de transporte, o protocolo OPC-UA pode utilizar tanto TCP, como HTTP, dependendo do formato de mensagem.

- **Formato Binário + TCP**: Neste formato os dados são colocados num vetor de bits. Este formato é utilizado por dispositivos com menos recursos e com o objetivo de atingir uma elevada performance, pois oferece menos custos computacionais no que toca à codificação/descodificação. Para este tipo de formato é utilizado o TCP, pois são mensagens mais simples e com as quais se pretende ter uma boa performance.

- **Formato XML + HTTP/SOAP**: Este formato de dados é o mais utilizado por aplicações presente numa camada superior, pois é fácil interpretar os dados apesar de ser computacionalmente mais complexo e consequentemente ter menor performance que o formato binário. Os dados ficam organizados através de *tags*, o que facilita a sua interpretação. Neste caso é utilizado HTTP, pois este suporta mensagens do tipo SOAP que permitem enviar ficheiros XML.

A utilização de OPC-UA a nível industrial está relacionada com a baixa latência imposta às mensagens, bem como o suporte na integração deste protocolo em PLC's. Desta forma, é utilizado em todo o tipo de controlo industrial, como é exemplo a implementação efetuada por Mizuya et al. [19]. Nesta implementação, o principal objetivo era controlar um manipulador robótico designado por SCARA (*Selective Compilance Assembly Robot Arm*). Para isso, inicialmente utilizou-se uma *Raspberry Pi* como *gateway* e um computador como servidor, sendo que posteriormente o controlo passou a ser efetuado por um PLC. Como protocolos, utilizaram-se o OPC-UA para o controlo e o MQTT para a monitorização.

Como foi referido anteriormente o OPC-UA, ao contrário das versões mais antigas, suporta a modelação de dispositivos, o que permite à camada de software superior abstrair-se da complexidade destes dispositivos. Para isso utiliza-se um paradigma orientado a objetos, sendo que este objetos são constituídos pelas seguintes componentes:

- **Variáveis**: As variáveis estão divididas em dois grupos, as variáveis de dados e as que contém propriedades, sendo que as primeiras contém os valores medidos pelo hardware (dinâmicas), e as últimas apresentam as características que descrevem o dispositivo.
- **Métodos**: Os métodos representam as funcionalidades que o dispositivo (servidor) pode efetuar, o cliente pode aceder a essas funcionalidades invocando os métodos correspondentes.
- **Eventos**: No caso dos eventos, estes são emitidos quando uma variável medida é alterada, todos os clientes com interesse nesse variável receberão esse evento.
- **Vistas**: Da mesma maneira que o SQL permite organizar dados, as vistas têm esse mesmo objetivo.
- **Referências**: As referências permitem ligar os objetos entre si, assim um cliente pode mais facilmente aceder ao objeto que mais lhe interesse.

2.1.5 Comparação dos protocolos

Em suma, a grande maioria dos protocolos descritos anteriormente, permite uma comunicação sem grandes latências, fiável e segura, sendo que a grande diferença passa pelo tipo de aplicação para o qual vão ser utilizados. Como representado na Tabela 2.1, os protocolos com *headers* menores e que utilizam UDP na camada de transporte, estão direcionados para redes e dispositivos com poucos recursos computacionais, sendo de destacar o CoAP neste particular.

Na Figura - 2.7, está representada a distribuição dos protocolos consoante a sua aplicação. Salienta-se a versatilidade do CoAP, pois pode ser utilizado tanto para a comunicação com os dispositivos IoT (esquerda da IoT *gateway*), bem como para a comunicação com *clouds* (direita da IoT *gateway*). Quanto ao resto de protocolos é de salientar que o OPC-UA é o melhor para redes locais industriais, e que para a comunicação com *clouds* o mais utilizado é o MQTT, pois

Tabela 2.1: Comparação dos protocolos.

Protocolo	***Header* (bytes)**	**QoS**	**Partilha de Serviços**	**Transporte**
MQTT	2	✓	✗	TCP
CoAP	4	✓	✓	UDP
AMQP	7	✓	✗	TCP
OPC-UA	-	✗	✓	TCP,HTTP

o AMQP está mais direcionado para *clouds* empresariais, como por exemplo bancos e grandes empresas.

Do ponto, de vista de performance, Chaudhary et al. [1] avaliaram 3 dos principais protocolos, neste caso MQTT, AMQP e CoAP, em três métricas principais, nomeadamente o *overhead* de pacotes, a taxa de envio de mensagens e a utilização de largura de banda. Numa outra comparação, realizada por Naik [20], foram utilizadas métricas, como a potência consumida pelos dispositivos, a latência, a segurança, a taxa de utilização em redes IoT e a fiabilidade.

- ***Overhead* de pacotes**: Este parâmetro de avaliação consiste no número de pacotes gerados pela rede para assegurar o envio de mensagens. Como era de esperar, os protocolos que utilizam TCP como transporte (MQTT e AMQP), geraram um número muito maior de pacotes para assegurar a entrega, comparativamente com o CoAP que utiliza UDP como transporte.

- **Taxa de envio de mensagens**: Quando são avaliados, relativamente ao número de mensagens enviadas por unidade de tempo, o protocolo que permite utilizar uma maior taxa é o MQTT.

- **Largura de banda**: Este campo, está relacionado com a utilização da largura de banda, por cada um dos protocolos, ou seja a velocidade de rede necessária. Neste particular, destaca-se o MQTT, pois para um elevado número de mensagens ele utiliza grande parte da largura de banda disponível, enquanto que nas mesmas condições o CoAP é o que menos largura de banda utiliza.

- **Potência consumida**: Em termos de potência consumida pelos dispositivos onde estão a operar os protocolos, como era de esperar, os que consomem menos potência são o CoAP e o MQTT, isto porque estão projetados para operarem em dispositivos com poucos recursos computacionais. Quanto ao AMQP, este já contém mais mecanismos tanto de fiabilidade

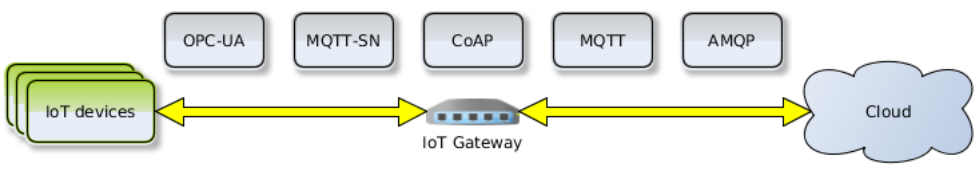

Figura 2.7: Disposição dos protocolos desde o dispositivo IoT até á *cloud*

como de segurança, desta forma é necessário a utilização de dispositivos com maior poder computacional.

- **Latência**: Como foi referido anteriormente, a utilização de UDP na camada de transporte, por parte do CoAP permite-lhe a utilização de uma menor largura de banda, o que implica uma menor latência na entrega das mensagens. Quanto ao AMQP, este é o que maior latência apresenta, devido a ser mais complexo que os outros.

- **Segurança**: No que diz respeito à segurança, destaca-se o AMQP, daí ser a escolha preferida de grandes empresas, para a implementação de *clouds*. O que apresenta piores mecanismos de segurança é o MQTT, pois apenas possui um mecanismo de autenticação, para os clientes.

- **Utilização em redes IoT**: A nível de utilização por parte de projetos, o MQTT é o que tem maior aceitação por parte das empresas, sendo utilizado por organizações como a IBM, a *Cisco*, a *Red Hat* e a *Amazon Web Services*. Quanto ao CoAP como é o mais recente, é também o menos utilizado, no entanto nos últimos anos a sua utilização tem vindo a aumentar.

- **Fiabilidade**: Relativamente à fiabilidade na entrega das mensagens, os protocolos que utilizam TCP apresentam melhor índice de entrega de mensagens, pelo que neste particular os que apresentam melhor fiabilidade são o MQTT e o AMQP.

Do ponto de vista industrial, o protocolo mais adequado é o OPC-UA, pois este protocolo foi projetado com o propósito de fazer a comunicação em aplicações industrias. Pfrommer and Palm [25] efetuaram um implementação do protocolo OPC-UA juntamente com o protocolo REST, sendo que esta implementação obteve excelentes resultados (comunicações na ordem dos poucos milissegundos). Para isso, adaptaram o protocolo OPC-UA para utilizar como camada de transporte o UDP ao invés do TCP, assim desta forma obtiveram pior fiabilidade mas melhor desempenho.

Outra implementação no âmbito industrial foi realizada, por Iglesias-Urkia et al. [11], onde fizeram uma comparação entre o CoAP e o MQTT, tanto numa fase de controlo como de monitorização. Desta forma, implementaram um sistema composto tanto por sensores como por atuadores, que comunicavam com um controlador (*Raspberry Pi*), através de um *broker* (*Raspberry Pi*). A nível de resultados foram avaliados 3 parâmetros, o *overhead* de pacotes, a escalabilidade, ou seja, a capacidade de ir acrescentando novos dispositivos à rede, e a latência no envio de pacotes. Relativamente ao *overhead* de pacotes, os autores chegaram à mesma conclusão referida anteriormente, em que o CoAP, por utilizar UDP como transporte, gera menos *overhead* de pacotes, em relação ao MQTT. Quanto a escalabilidade, para um elevado número de dispositivos presentes na rede, o CoAP tende a ter melhor performance em relação ao MQTT, pois permite a utilização de *multicast*, seja em redes locais IPv4 ou IPv6. Desta forma, economiza-se um elevado número de recursos, pois não é necessário uma conexão para cada dispositivo, como acontece no MQTT. A maior diferença entre os dois protocolos refere-se à latência entre ambos, em que o

CoAP apresenta valores na ordem dos microssegundos, enquanto que o MQTT apresenta na ordem dos milissegundos. A conclusão deste artigo foi que para aplicações industriais, em que é necessário um controlo em tempo real, o CoAP cumpre melhor essa tarefa em relação ao MQTT.

2.2 Plataformas IoT

Uma plataforma IoT caracteriza-se por efetuar a gestão de dispositivos IoT, bem como de plataformas externas, isto é, suporta a descoberta de serviços, bem como a normalização de comunicações entre diferentes dispositivos IoT, que utilizam diferentes protocolos de comunicação, como os identificados na secção anterior. A implementação de um CPPS é facilitada com a utilização de plataformas IoT. De seguida identificam-se algumas das plataformas IoT principais.

2.2.1 FIWARE

O FIWARE é uma plataforma IoT que resultou de um projeto europeu, que visa normalizar as comunicações em sistemas IoT. Com o objetivo de incentivar a sua utilização por parte do mercado das aplicações IoT, esta plataforma IoT é disponibilizada de forma gratuita.

A componente principal desta plataforma IoT é o *broker* de contexto, pois é este que permite o registo de entidades, o armazenamento dos dados, a subscrição de atributos, etc. O *broker* de contexto comunica através do protocolo HTTP, utilizando o formato de dados JSON para estruturar a mensagem. Assim, para tornar esta plataforma mais versátil a nível de protocolos, são utilizados agentes IoT, para fazerem a tradução de MQTT ou CoAP, para HTTP. Desta forma, a informação chega proveniente do dispositivos IoT, presentes nas redes locais ao agente, onde é tratada consoante o protocolo utilizado na comunicação.

A implementação do FIWARE está dividida em diferentes componentes denominadas de *generic enablers* (GEs). Desta forma, cada um destes GEs está responsável por uma funcionalidade, seja o processamento de grande quantidade de dados, a segurança em redes ou até mesmo processamento de dados baseado em inteligência artificial. De todos os GEs existentes, de seguida vão ser descritos os responsáveis pela implementação de um sistema IoT:

- **IoT *Discovery***: No FIWARE, a descoberta de serviços está disponível através do IoT *Discovery*, que permite verificar quais os tópicos que estão ou não disponíveis, sendo utilizado pelas diferentes componentes ou através de uma interface gráfica. A descoberta de serviços pode ser efetuada utilizando diferentes mecanismos, nomeadamente geográficos, probabilísticos, ou semânticos.

- **IoT *broker***: O papel do IoT *broker* na *cloud* é de virtualização dos dispositivos presentes na rede, de maneira a facilitar o acesso por parte dos GEs da camada superior, aos dados provenientes de cada dispositivo. Assim este GEs serve de intermediário na comunicação entre os agentes IoT e o *broker* de contexto.

- ***Data Handling***: Os diferentes dispositivos presentes geram uma grande quantidade de dados heterogéneos. Este fenómeno é mais conhecido como Big Data. Se esta informação não for processada, filtrada e agregada antes de ser enviada para o servidor (*cloud*), vai causar uma sobrecarga deste, pondo em causa o funcionamento do sistema. Para evitar esta situação, foi desenvolvida uma componente ao nível da rede local (*Data Handling*), responsável por fazer este primeiro processamento da informação. Esta componente é normalmente utilizada em sistemas que requerem um controlo em tempo real, como linhas de montagem/fabrico, controlo de tráfego aéreo e redes de sensores, pois esta componente situa-se mais próxima da fonte de informação.

- ***Protocol Adapter***: O *Protocol Adapter* lida com o tráfego, entre o IoT *gateway* e os dispositivos, estes dispositivos por norma não são compatíveis com a rede IoT, por isso é necessário adaptar a comunicação de modo a serem integrados. Desta forma, está suportada a comunicação com dispositivos que utilizem a pilha protocolar IP (IPv4 ou IPv6).

Para a comunicação entre componentes, é utilizada uma *interface* designada por NGSI, também desenvolvida no âmbito do projeto FIWARE. Esta *interface* utiliza como protocolo de comunicação o HTTP e utiliza ficheiros do tipo XML para troca de informação. A Figura - 2.8 representa uma possível arquitetura para o sistema FIWARE.

A arquitetura para o sistema FIWARE é composta pelas componentes descritas anteriormente. Algumas das componentes presentes nesta arquitetura (IoT *broker*, IoT *Discovery*) estão ainda em fase de teste/desenvolvimento, desta forma em artigos, a maioria dos autores adota arquiteturas mais simples. Desta forma na maioria das implementações desta plataforma IoT, os autores tendem a utilizar apenas o agente IoT e o *broker* de contexto *Orion*.

2.2.1.1 Agentes IoT

Os agentes IoT são responsáveis por três funções fundamentais no servidor FIWARE, assim como representado na Figura - 2.9, nomeadamente: 1) conectar dispositivos físicos com o servidor; 2) servir como *handler*, para reencaminhar os pedidos efetuados pelos dispositivos para o *broker* de contexto; 3) permitir a utilização do IoT *manager*, para gerir os diferentes agentes IoT, sendo que o IoT *manager* está encarregue de ajudar na configuração, operação e monitorização. De momento esta GE permite comunicar utilizando três protocolos, o HTTP, o CoAP, e o MQTT, além de ser disponibilizada uma biblioteca que permite desenvolver, caso necessário, uma aplicação que utilize outro protocolo de comunicação.

Os agentes IoT traduzem a informação proveniente dos dispositivos da rede em entidades de contexto, servindo estes agentes como produtores para o *broker* de contexto. O agente IoT deve funcionar também como provedor de contexto caso o *broker* de contexto necessite de enviar comandos ou informações para um dispositivo.

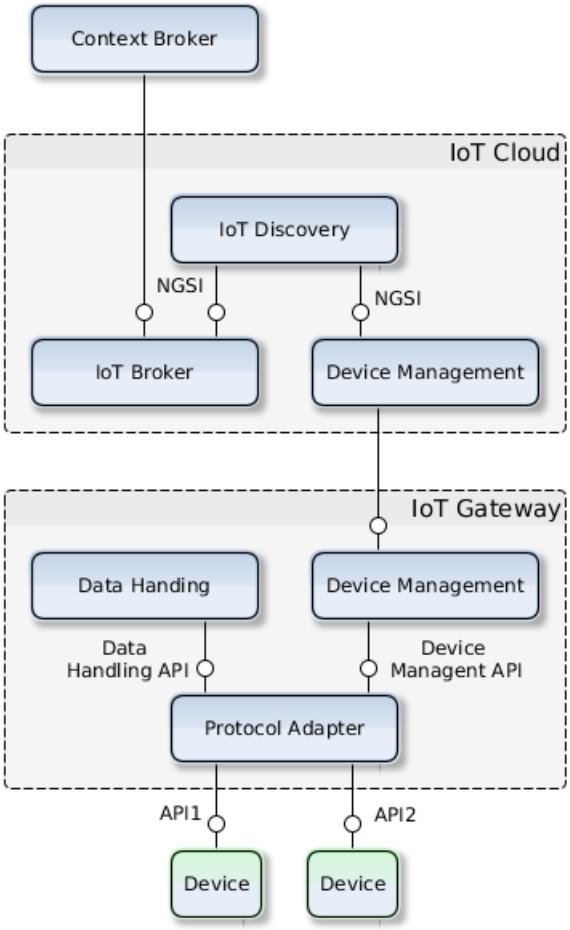

Figura 2.8: Arquitetura FIWARE.

2.2.1.2 Agente IoT: Ultralight

O agente IoT do tipo *Ultralight* utiliza como protocolo o HTTP ou o MQTT. Como estes protocolos utilizam o TCP na camada de transporte, tem como consequência uma maior fiabilidade na entrega das mensagens apesar de maior *overhead* de pacotes e atraso na entrega. A grande diferença entre este agente e o agente JSON está relacionada com o formato das mensagens, pois neste agente o formato utilizado é mais compacto que o JSON, apesar de ser mais difícil de interpretar pelo código. Desta forma, este agente permite aos dispositivos com quem comunica as seguintes funcionalidades:

- **Criação de serviço**: A criação do serviço é realizada por um administrador. Esta funcionalidade permite agrupar os dispositivos de uma forma mais organizada. Para isso, é enviado como argumento o nome do serviço e uma chave de autenticação (*API-key*).

- **Registo/Criação de dispositivos**: Antes de começar a enviar/receber informação, o dispositivo tem de se registar. Para isso, o dispositivo envia um pedido para o agente IoT, que

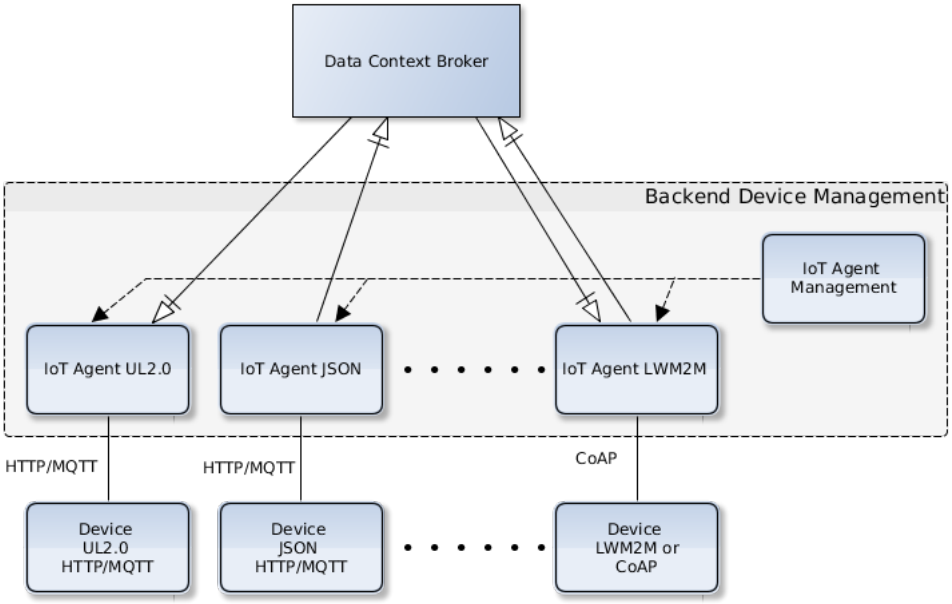

Figura 2.9: Arquitetura do agente IoT.

deve incluir tanto o ID do dispositivo como o da entidade, o tipo de entidade, os atributos (estáticos ou dinâmicos) e a lista de comandos que o dispositivo suporta.

- **Observação do dispositivo**: Após o dispositivo estar registado na *cloud*, este pode começar a enviar-lhe informação, sendo que estas mensagens devem conter também o ID do dispositivo e a *API-key*.

- **Envio de comandos**: Os comandos podem ser enviados para o dispositivo de 2 maneiras diferentes:

  - ***Pull***: Desta maneira o agente envia, de maneira síncrona, o comando para o dispositivo, e espera pela resposta do mesmo.

  - ***Push***: Neste caso quem toma a iniciativa na transmissão é o dispositivo, ao qual lhe é respondido com um lista de comandos existentes no agente IoT.

2.2.1.3 Agente IoT: JSON

Relativamente ao agente JSON, comparativamente com o agente UL, tanto os métodos utilizados na comunicação como os protocolos suportados são os mesmos, sendo a única diferença o formato das mensagens. Desta forma, este agente adota o JSON como formato predefinido das suas mensagens. Uma vantagem da utilização deste tipo de formato é o facto da maioria das aplicações Web suportar o formato JSON, daí introduzir maior robustez ao sistema e maior facilidade em adicionar novas componentes á rede.

2.2.1.4 Agente IoT: LWM2M

Quanto ao agente *lightweight* M2M, este é caracterizado por utilizar CoAP como protocolo. Como foi referido em 2.1.2, o CoAP utiliza UDP na camada de transporte, assim este agente IoT adequa-se a dispositivos com maiores limitações ao nível dos recursos computacionais, dai possuir funcionalidades diferentes do UL2.0. Neste caso, as funcionalidades são:

- **Criação de serviços**: A criação de serviços cobre a mesma funcionalidade que foi descrita acima, diferindo apenas nos parâmetros que são enviados para o agente IoT. Assim os parâmetros enviados são a chave de autenticação e os recursos que permitem aceder ao nó da rede onde está o dispositivo LWM2M.

- **Registo de dispositivos**: Este caso de utilização tem o mesmo fim que foi referido anteriormente, sendo apenas distintos tanto os parâmetros como a resposta do agente IoT. Os parâmetros passam a ser o *endpoint* (endereço IP e porta) em que se situa este dispositivo, e os objetos que pretende registar. A resposta do agente IoT é neste caso uma lista dos objetos que ficaram ativos e vão ser monitorizados.

- **Observação rápida de dispositivo**: A observação rápida de um dispositivo tem como iniciador da comunicação o agente IoT, que envia um pedido de leitura ou escrita ao cliente LWM2M. Pode ser equiparada a uma observação do tipo *pull*.

- **Observação ativa de dispositivo**: Quanto à observação ativa, esta é despoletada pelo cliente LWM2M, que envia uma notificação como o valor medido para o agente IoT. Neste caso pode ser equiparada a uma observação do tipo *push*.

- **Enviar comandos para o dispositivo**: A tarefa de enviar comandos neste agente IoT é mais simplificada, pois o agente IoT apenas envia o comando que pretende executar e o cliente LWM2M retorna o seu estado.

2.2.1.5 *Orion Context Broker*

O *broker* de contexto está presente no topo da arquitetura de uma *cloud* FIWARE. Desta forma, esta componente utiliza as ferramentas disponibilizadas pelos GEs presentes nas camadas mais abaixo, de maneira a fazer um boa gestão do conteúdo recebido pelos publicadores e da informação partilhada com os subscritores. Este *broker* permite fazer também uma partilha inteligente dos conteúdos com os seus subscritores, ou seja, para além de partilhar com eles a informação pedida, é partilhada também a informação relacionada com o contexto do pedido. Por exemplo, se for pedida a informação de um sensor de pressão na máquina A, o *broker* de contexto partilha essa informação com quem fez o pedido em primeiro lugar, como também a relacionada com os outros sensores de máquina. Para a implementação desta funcionalidade é crucial a utilização dos mecanismo de descoberta de serviços.

A arquitetura está representada na Figura - 2.10, onde está explicito que o *broker* de contexto faz a intermediação entre os publicadores e os subscritores. Além de fazer o controlo de fluxo, agrega a informação consoante as subscrições, entre outros.

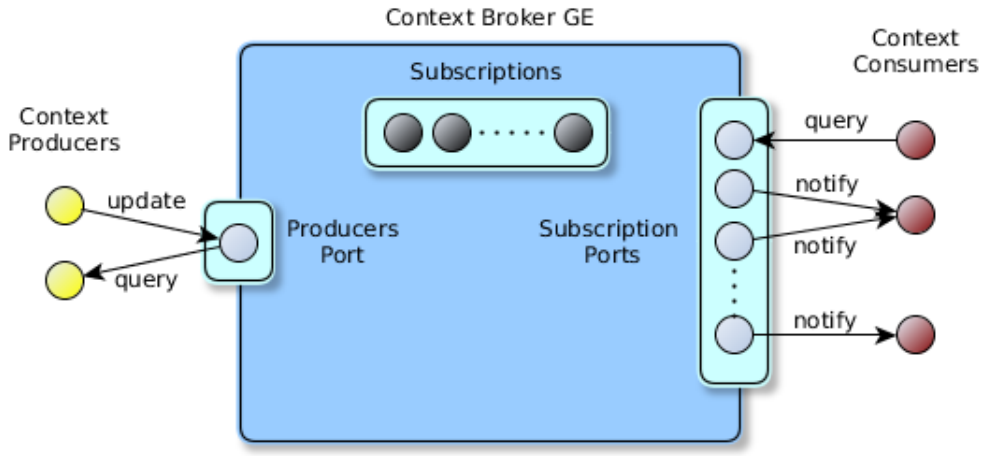

Figura 2.10: Arquitetura do *broker* de contexto.

A partilha de informação entre publicadores e subscritores possui três características importantes ao nível do:

- **Conhecimento**: Não é necessário conhecimento mútuo entre os dispositivos que efetuam a publicação da informação e os que a consomem. O conhecimento mútuo significa que não é necessário um dispositivo saber as características físicas do outro (por exemplo, endereço IP, porta, etc).

- **Sincronização**: Não é necessário nenhuma sincronização entre os intervenientes, ou seja, os publicadores não são bloqueados enquanto produzem a informação, sendo que os subscritores também podem consumir a informação de forma assíncrona.

Estas características são cruciais, pois permitem diminuir as dependências entre a produção e o consumo de informação, ou seja aumenta a tolerância a falhas do sistema distribuído e permite a utilização de componentes com recursos mais limitados. Os publicadores e subscritores interagem com o *broker* de contexto, através dos seguintes métodos:

- ***query***: Um pedido de *query*, o qual é despoletado pelo subscritor, consiste num pedido de atualização de um atributo.

- ***update***: Este método permite a um publicador partilhar a seu estado com os subscritores interessados. Este pedido é despoletado pelo publicador, sendo que quando o *broker* de contexto recebe esta informação, notifica os subscritores interessados.

- ***notify***: No caso deste método, é possível notificar os subscritores, da atualização de um atributo por parte do publicador.

- ***discover***: Este método permite a um dispositivo descobrir quais as entidades que estão relacionadas com o tópico a que pertencem.

- ***subscribe***: Este método permite a um dispositivo subscrever um determinado tópico.

- ***register***: Este método permite um publicador registar os tópicos dos quais mais tarde vai difundir informação.

2.2.1.6 Base de Dados

Para persistência de informação, o *broker* de contexto *Orion* utiliza como base de dados a *MongoDB*. Esta base de dados é não-relacional (NoSQL), ou seja, os dados não obedecem a uma estrutura fixa. Ao contrário, os dados são guardados em documentos, que são organizados em *collections*, como representado na Figura - 2.11, onde estão salientes as diferenças entre uma base de dados relacional, e uma base de dados documental.

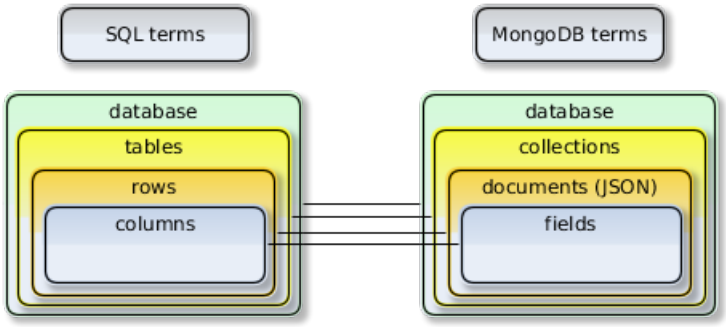

Figura 2.11: Comparação do modelo relacional com o modelo utilizado pela *MongoDB*.

Em comparação dois dois tipos de base de dados, destaca-se a substituição das tabelas por *collections*, das linhas por documentos e das colunas por atributos. Os documentos utilizados para guardar os dados são ficheiros do tipo JSON, como representado na Figura - 2.12, onde estão apresentadas duas *collections*, sendo que cada uma tem um certo número de documentos (JSON) associados.

A utilização de ficheiros do tipo JSON aumenta a flexibilidade da estrutura, pois permite modificar e adicionar campos, sem alterar drasticamente a estrutura ou implicar a migração de dados. Relativamente a aplicações para este tipo de base de dados, a sua utilização está a aumentar na área das aplicações IoT, pois permite adicionar novas máquinas, de forma a aumentar a capacidade de armazenamento. Esta funcionalidade é muito útil pois os sistemas IoT produzem uma grande quantidade de dados, e à medida que os sistemas vão ficando mais complexos é necessário aumentar a capacidade de armazenamento. Desta forma, as vantagens das bases de dados documentais em relação às relacionais são:

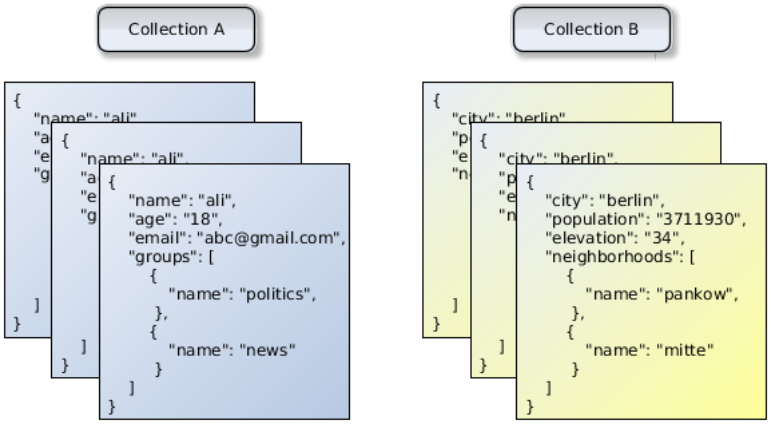

Figura 2.12: Estrutura de um documento e de uma *collection*.

- **Escalabilidade**: A base de dados *MongoDB*, ao ser do tipo documental, oferece maior flexibilidade e escalabilidade, pois os atributos de um documento podem ser modificados sem condicionar outros documentos.

- **Performance**: A nível de performance, as pesquisas são muito mais rápidas, pois através de uma consulta, obtêm-se todos os dados relacionados com o documento.

- **Simplicidade**: Comparativamente às bases de dados tradicionais, a *MongoDB* permite efetuar consultas de uma forma mais simples, pois não é necessário utilizar funções complexas (*join*, etc).

- **Armazenamento**: Como foi referido anteriormente, a *MongoDB* permite alocar armazenamento à medida que é necessário, isto através do aumento de máquinas. Relativamente às bases de dados relacionais constitui uma vantagem pois está muito mais simplificado este processo.

2.2.1.7 Aplicações

A nível de aplicações reais, esta plataforma IoT já foi utilizada para todo o tipo de aplicações. Por exemplo, o FIWARE foi utilizado em sistemas de monitorização de pacientes, tendo sido implementado por Fazio et al. [5], como forma do médico ter acesso aos dados de cada paciente. Foi também a base dum sistema de monitorização de campos agrícolas, implementado por López-Riquelme et al. [17], com o objetivo de implantar um conjunto de técnicas de precisão na gestão de recursos como a água. Para além disso, também foi utilizado numa *cloud*, que tinha como seus produtores sensores que permitiam o reconhecimento de diferentes tipos de gestos que o utilizador estivesse a fazer, juntamente com esta implementação, Preventis et al. [26] fizeram também uma análise dos atrasos presentes no sistema estando estes situados na ordem das dezenas de milissegundos.

2.2.2 *Amazon Web Services* IoT

A AWS IoT é uma plataforma desenvolvida pela *Amazon*, uma das maiores empresas de soluções digitais da atualidade. Esta plataforma, à semelhança das outras, permite implementar tanto uma *cloud* IoT, como configurar as IoT *gateways* se necessário. Porém, como é uma solução proprietária, o código fonte não é disponibilizado, o que constitui uma grande desvantagem em comparação com outras soluções de código aberto. A publicação/subscrição pode ser efetua, por parte dos dispositivos através de dois protocolos, nomeadamente o MQTT e o HTTP.

Guth et al. [9] efetuaram uma análise da AWS IoT e do FIWARE, em que descrevem a arquitetura do AWS IoT, que está representada na Figura - 2.13.

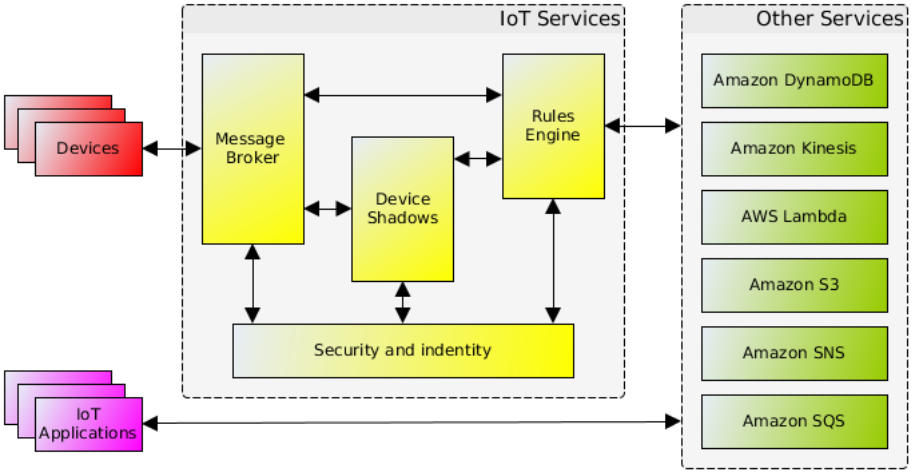

Figura 2.13: Arquitetura presente na AWS IoT.

A arquitetura do AWS IoT é composta pelas seguintes componentes:

- **Message Broker**: Componente responsável por receber/enviar as publicações/subscrições de cada um dos dispositivos presentes na rede.
- **Rules engine**: Esta componente disponibiliza mecanismos de processamento e integração com os outros serviços AWS. Desta forma, esta componente seleciona o *payload* existente na mensagem, através de uma linguagem baseada em SQL, sendo que posteriormente disponibiliza esse mesmo conteúdo aos outros serviços.
- **Device Shadows**: Esta componente consiste numa réplica digital dos dispositivos presentes na rede, desta forma esta réplica contém os atributos do dispositivo, os certificados utilizados na autenticação, as operações/funcionalidades que o dispositivo disponibiliza, etc.
- **Security and identity**: Esta componente está encarregue de implementar a parte de segurança. Para isso, os dispositivos necessitam de efetuar uma autenticação tanto para publicarem como para subscreverem tópicos. Além disso, permite que os dispositivos sejam agrupados consoante o registo que efetuaram.

Uma implementação foi realizada por Tärneberg et al. [28] e consiste num sistema inteligente para controlo de sinais de trafico, onde o principal objetivo é de manter a pontualidade dos transportes públicos numa cidade inteligente. Nesta implementação foram utilizados os serviços AWS responsáveis por desenvolver um sistema IoT, bem como as componentes responsáveis por análise e tratamento de dados, *Lambda*, *DynamoDB* e *Kinesis*.

- **Lambda**: Consiste na plataforma onde são implementadas as funções pretendidas. A título de exemplo, Tärneberg et al. [28] usaram esta plataforma para implementar as funções para avaliar a pontualidade dos autocarros.
- **DynamoDB**: Esta componente consiste numa base de dados NO-SQL, com um nível baixo de latência, que permite guardar os dados provenientes dos dispositivos.
- **Kinesis**: Esta componente consiste num *buffer* de dados que permite guardar e agregar os dados consoante as suas características.

2.2.3 *Microsoft Azure*

Relativamente ao *Microsoft Azure*, é uma plataforma IoT desenvolvida pela Microsoft, em que a estrutura em que está assente é em tudo semelhante às descritas anteriormente. Neste caso é composta por uma unidade de tratamento e processamento de dados (*cloud*), e por diversos dispositivos que podem estar ligados a esta *cloud* através de IoT *gateways*, ou diretamente caso tenham capacidade de utilizar um IP. Esta arquitetura está representada na Figura - 2.14.

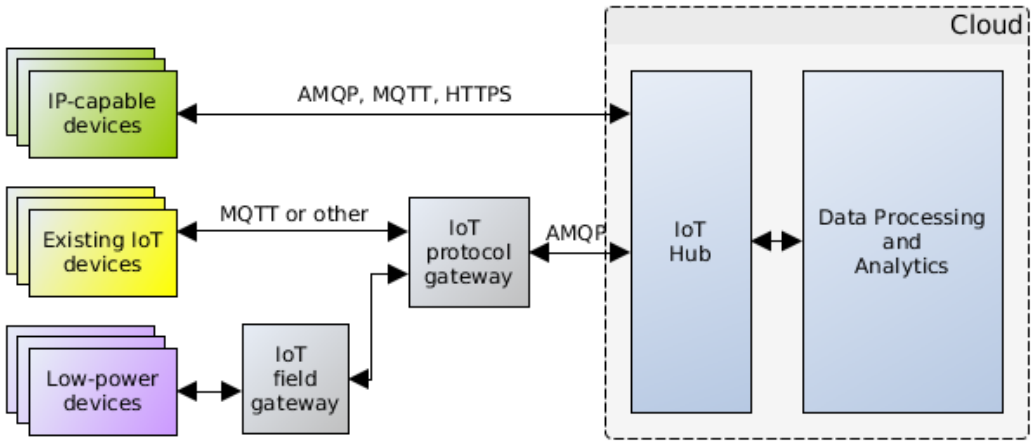

Figura 2.14: Arquitetura presente no *Microsoft Azure*.

Nesta arquitetura estão explícitos os protocolos de comunicação que a *cloud* suporta, nomeadamente o AMQP, o CoAP e o HTTPS. Destaca-se que a comunicação desde o dispositivo até à IoT *gateway* pode ser efetuada num protocolo qualquer, desde que seja feita uma tradução, nesta mesma *gateway*, para um dos três protocolos referidos anteriormente, como descrito por Forsstrom and Jennehag [6], que apresenta uma implementação do *Microsoft Azure* como servidor de

cloud, e em que a comunicação até à *gateway* é efetuada através de OPC-UA. Este sistema IoT tinha como função a monitorização de 1500 sensores presentes numa fábrica, sendo que 600 deles atualizam o seu valor a cada segundo, assim como representado na Figura - 2.15.

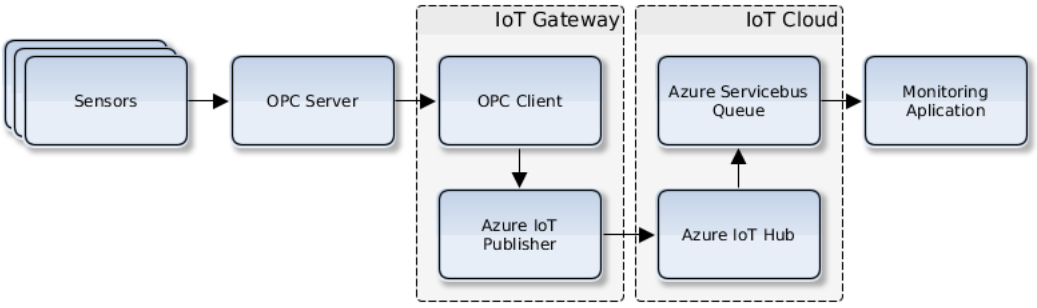

Figura 2.15: Implementação conjunta de *Microsoft Azure* e OPC-UA.

Do ponto de vista dos resultados, esta implementação obteve tempos de resposta, no intervalo dos 1-3 milissegundos, do sensor até à *gateway*, e no intervalo dos 700-800 milissegundos, desde a *gateway* até à plataforma de monitorização. Tendo em conta que o objetivo é monitorizar uma quantidade enorme de sensores, estes resultados são promissores. A partir destes valores conclui-se que a utilização de OPC-UA na rede local da fábrica é um sucesso, enquanto que a utilização do *Microsoft Azure* no resto do sistema podia perfeitamente ser substituída por qualquer uma das plataformas IoT descritas anteriormente.

Outra desvantagem desta plataforma IoT é o custo que têm os serviços de armazenamento de mensagens da *Microsoft Azure*. Por exemplo, no projeto descrito anteriormente, são geradas cerca de 52 milhões de mensagens por dia, a utilização dos serviços de armazenamento da Microsoft é gratuita até 8000 mensagens por dia, sendo que neste caso esse serviço custou-lhes 5000$ mensais.

2.2.4 Outros plataformas IoT

Além das plataformas IoT descritos anteriormente (FIWARE, *Microsoft Azure* e AWS IoT), existem outros, que também constituem uma boa opção para plataformas IoT industriais. Assim, de seguida vão ser descritas de forma sucinta algumas das plataformas IoT, que falta referir nesta revisão bibliográfica, sendo estas o *SiteWhere*, o *Linksmart* e o *Konker*.

Começando pela descrição do *SiteWhere*, este é uma plataforma IoT disponibilizada gratuitamente no *github* [16], sendo este desenvolvido em Java pela empresa *SiteWhere LLC*. Este é um projeto bastante maduro, com constantes atualizações e já apresenta um volume de 250.000 linhas de código. Em termos de arquitetura apresenta uma arquitetura centralizada baseada num servidor, este servidor é capaz tanto de receber dados através dos dispositivos, bem como de enviar comandos para estes mesmos, a comunicação com os dispositivos funciona através dos protocolos MQTT ou AMQP. De forma a integrar plataformas externas foi desenvolvida um API REST, que permite comunicar com outras plataformas que usem esta mesma API, como por exemplo a página de administração do *SiteWhere*. Em termos de plataformas capazes de interagir com a plataforma

IoT do *SiteWhere*, já foram integradas o *Apache Solr*, o *Twilio Cloud Communication*, o *Microsoft Azure EventHub* e o *MuleSoft AnyPoint Platform*. Para finalizar em termos, de armazenamento de grande volume de dados, o *SiteWhere* utiliza a base de dados não-relacional *MongoDB* ou a *Apache Hbase*. Para mais informação relacionada com este protocolo basta consultar a página onde se encontra disponibilizada a documentação [15].

Quanto ao *Linksmart*, este é uma plataforma IoT desenvolvida pela *Atlassian Corporation*. Em termos de arquitetura esta plataforma está dividida em diferentes componentes responsáveis pelas mais diversas funções como, a virtualização e integração dos dispositivos, a gestão do grande volume de dados gerados pelos dispositivos, através de métodos de *machine learning*, a disponibilização de serviços, a visualização dos dados e dos processos, a segurança dos dados e por fim a integração de outras plataformas. A componente responsável pela integração e virtualização dos dispositivos funciona ao nível da *gateway*, desta forma disponibiliza um ponto de contacto para que os dispositivos presentes nesta sub-rede, interatuem com os serviços fornecidos pelas outras componentes associadas ao *Linksmart*. Em termos, de integração com outras plataformas, é possível associar o *Bitbucket*, o *Jira*, o *Bamboo* e o *Nexus*. Relativamente, à documentação esta pode ser encontrada online no site [4].

Por fim, o *Konker*, é uma plataforma menos popular que as anteriores, usando esta uma arquitetura centralizada, baseada numa *cloud* prestadora de serviços. Em termos de custos, inicialmente é disponibilizado um plano gratuito, mas com limitação de dispositivos, sendo que se for necessário aumentar o número de dispositivos tem de se optar por planos pagos. Em termos, de arquitetura, utiliza um modelo bastante simplificado, em que apenas existe a *cloud* centralizada e os dispositivos que produzem ou consomem dados. Quanto aos protocolos suportados, os dispositivos podem comunicar com a *cloud* através de HTTP/REST ou MQTT. Quanto, à documentação referente a esta plataforma IoT, esta pode ser encontrada no site [14].

2.2.5 Comparação das plataformas

Baseando-se nas descrições das plataformas IoT, em termos de funcionalidades como, a descoberta de serviços, a virtualização dos dispositivos, a autenticação dos dispositivos, etc, todos eles suportam essas funcionalidades. Na Tabela 2.2 representam-se as características em que diferem as plataformas IoT identificadas.

Tabela 2.2: Comparação dos plataformas IoT.

Plataforma IoT	**Protocolos Suportados**	***Open Source***	**Base de Dados**	**Licença**
FIWARE	MQTT,CoAP,REST	✓	mongoDB	GPL
AWS IoT	MQTT,HTTP	✗	dynamoDB	-
Microsoft Azure	AMQP,MQTT,HTTPS	✗	SQL,cosmosDB	-
SiteWhere	MQTT,AMQP,REST	✓	mongoDB	CPAL-1.0
Linksmart	REST	✓	mongoDB,influxDB	Apache 2.0
Konker	MQTT,REST	✓	mongoDB	Apache 2.0

Verifica-se que relativamente às bases de dados utilizadas, todas as plataformas IoT dão preferência às bases de dados não-relacionais (NoSQL), com exceção do *Microsoft Azure* que disponibiliza uma base de dados relacional. A utilização de bases de dados não-relacionais é uma vantagem nos sistemas IoT, pois são mais flexíveis, no que toca à alteração da estrutura dos dados. As plataformas IoT disponibilizadas em *open source* são mais vantajosas, pois estas disponibilizam tanto o código fonte como a documentação de forma gratuita, o que constitui uma enorme vantagem. Quanto aos protocolos utilizados, destaca-se a unanimidade da utilização do HTTP/-REST e MQTT entre as plataformas, o que vem a confirmar a grande popularidade do MQTT nas aplicações IoT, bem como a recorrente utilização de protocolos, normalmente utilizados em aplicações Web (HTTP/REST). Além disso, salienta-se que o único que suporta a utilização de CoAP, na comunicação entre a *cloud* e os dispositivos IoT ou *gateways*, é o FIWARE. Como foi referido anteriormente 2.1.2, o CoAP é um protocolo leve, que funciona perfeitamente em sistemas industriais.

Para complementar os estudo feito referente às plataformas, Guth et al. [9] descreve várias plataformas, que posteriormente são comparadas através de métricas qualitativas e quantitativas.

Relativamente às métricas qualitativas, são utilizados o número de atualizações da plataforma IoT, que permite inferir se se trata de um projeto promissor ou não, a popularidade, que está relacionada relacionada com o prestigio do produto e o número de protocolos suportados, que traduz a versatilidade e adaptabilidade da plataformas. Na comparação destes parâmetros, foram consideradas onze plataformas IoT, sendo que apenas cinco passaram à fase da comparação com parâmetros quantitativos, desta forma foram descartados o *Nitrogen*, o *OpenIoT*, *Nimbits* e o *Webinos*, por não apresentarem atualizações recentes, ou seja, aparentarem ser projetos abandonados ou com *clouds* desconectadas. A acrescentar a estas quatro plataformas descartadas foram adicionados o *Devicehive* e o *Kaa*, estes dois devido a problemas técnicos, o primeiro por perder os dados quando se efetuava um reinicio do sistema, e o segundo por não ser possível obter dados da plataforma sem desenvolver a sua própria aplicação.

Assim, os autores comparam o *Orion* (FIWARE), o *SiteWhere*, o *InatelPlat*, o *Konker* e o *Linksmart*. As métricas quantitativas utilizadas foram o tamanho dos pacotes, a percentagem de erro que ocorre na receção da informação proveniente dos dispositivos e o tempo de resposta, tanto o tempo de resposta como o tamanho dos pacotes permitem inferir sobre o constrangimento da rede. No primeiro dos teste, foi medido o tamanho dos pacotes enviados pela plataforma IoT em resposta a um pedido REST, com 1, 10 e 100 parâmetros, por parte de um dispositivo. Neste particular, a plataforma IoT com melhores resultados foi o *Konker*, enquanto que o *Orion* apresentou as mensagens com maior número de bytes. É de salientar que o *Konker* e o *Linksmart* apresentam sempre o mesmo tamanho de mensagem independente do número de parâmetros utilizados, o que constitui uma vantagem do ponto de vista do congestionamento imposto à rede. No segundo teste efetuado foi avaliada a percentagem de erro de cada uma das plataformas, para isso foram utilizados 100, 1000, 5000 ou 10000 utilizadores (dispositivos), em que cada utilizador enviava mensagens com 1, 15 ou 100 parâmetros, este teste pode ser interpretado como sendo um teste de fadiga do sistema. Relativamente a resultados, os que se demonstraram melhor desempenho foram

o *Orion* e o *SiteWhere* com valores a rondar os 0% para todos os casos, enquanto o *Linksmart* e o *Konker* demonstraram pior desempenho nos casos de maior fadiga (maior número de utilizadores e parâmetros), apresentando o primeiro valores superiores aos 60%, e o segundo valores a rondar os 25%, ficando comprovado que para um grande volume de dados não são uma boa opção. Por último, foi realizado o teste do tempo de resposta das diferentes plataformas IoT, nos mesmos casos descritos do teste da percentagem de erro (utilizadores/parâmetros), sendo que no caso do tempo de resposta os que apresentaram melhores resultados foram o *SiteWhere* e o *Orion*. A nível global as plataformas IoT que demonstraram melhor desempenho foram o *SiteWhere* e o *Orion*, provando que são os mais estáveis e robustos quando se está a processar um grande volume de dados.

2.3 Arquiteturas IoT

Nos dias de hoje, com base em processamento de alto nível de informação disponíveis, empresas como a *Google*, o *Facebook* ou a *Amazon*, fazem dos dados a sua maior fonte de lucro, utilizando-os como forma de prever que tipo de anúncios devem mostrar a um certo perfil de utilizador, ou mesmo até para estudos estatísticos mais complexos. Por vezes, a utilização abusiva destes dados, causou certos escândalos ao nível da privacidade dos utilizados, levando à restrição no que toca à legislação que controla este sector. No setor industrial, os dados são igualmente relevantes, pois é a partir de informação gerada no chão de fábrica que se consegue tirar conclusões de alto nível sobre o estado dos processos de produção, de forma a proceder a melhorias contínuas.

2.3.1 *Industrial Data Space*

Recentemente, surgiu um novo conceito relacionado com a gestão dos dados, nomeadamente a *Industrial Data Space* (IDS), que consiste genericamente em conectar todas as entidades geradoras de dados presentes em diferentes sectores. Os sectores abordados vão desde a industria, ao governo ou a empresas de distribuição. Assim, com a partilha de dados entre estas diferentes entidades, é possível otimizar os respetivos processos. A arquitetura do IDS está representada na Figura - 2.16.

A nível estrutural, o IDS é modelado por uma arquitetura composta pelas seguintes componentes: 1) provedores/consumidores de dados; 2) *brokers*; 3) operadores de aplicações (*App Store*); 4) autoridades de supervisão. Relativamente aos operadores de aplicações, estes consistem numa variedade de serviços que são disponibilizados e permitem efetuar o tratamento/processamento de dados. Quanto as autoridades de supervisão, estas garantem que as componentes desenvolvidas para o IDS cumprem certos requisitos pré-definidos e as regras estipuladas. O *broker* apresentado na Figura - 2.16, difere do *broker* utilizado no protocolo MQTT pois, neste caso, o *broker* é utilizado como base de dados que permite o registo de novas componentes, e o armazenamento dos dados produzidos por estas. Para efetuar a conexão entre as diferentes componentes presentes na rede local, é utilizado um conector interno, enquanto que para efetuar a conexão com outras componentes através da Internet é utilizado o conector externo.

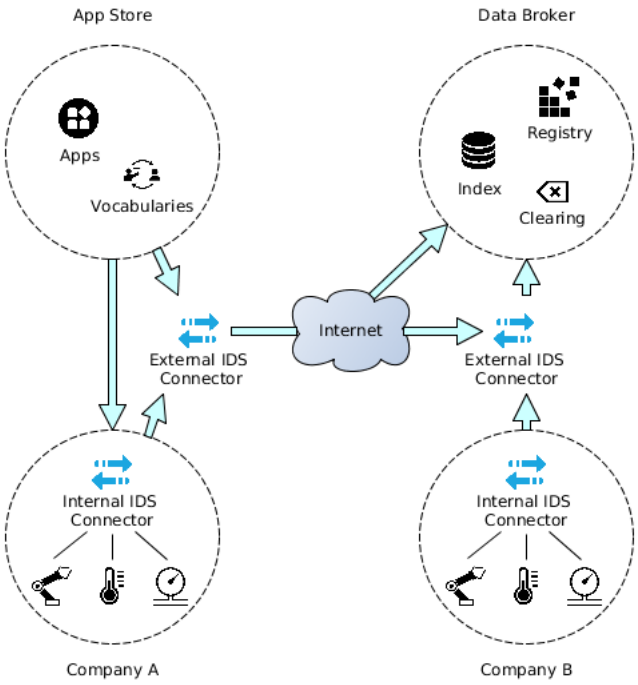

Figura 2.16: Arquitetura do IDS.

2.3.1.1 Aplicações

Quanto às diferentes aplicações do IDS, Otto et al. [22] apresenta alguns exemplos das aplicações reais do IDS. O primeiro exemplo apresentado é o caso de uma cadeia de abastecimento, onde a partilha de dados entre fornecedores, clientes e os serviços de logística permitem uma otimização de todo este processo. Em termos de dados utilizados seriam capturados os dados do GPS dos camiões, as informações do tráfego atual, os pedidos efetuados, os dados respetivos ao material e os dados correspondentes tanto aos fornecedores como aos clientes. A partir destes dados é possível fazer uma gestão em tempo real de todo o processo, estimando o staff a utilizar e reduzindo o risco associado à cadeia de abastecimento.

Uma outra aplicação desta arquitetura é na produção de produtos farmacêuticos. Neste caso, os intervenientes seriam os centros de investigação, as industrias farmacêuticas, as companhias de seguros, as fábricas de equipamentos médicos e os centros de saúde. Os dados utilizados por este modelo seriam os dados médicos pessoais, os estudos clínicos, os dados ambientais e os dados relacionados com os estudos de mercado. Com estes dados seria possível melhorar o processo de pesquisa e desenvolvimento de novos medicamentos e adaptar a produção ao consumo atual.

O caso das fábricas inteligentes também é referido, em que os intervenientes são os serviços de manutenção e os operadores de produção. Com base nestes intervenientes, é possível obter dados referentes à quantidade de produtos finalizados, ao estado atual dos operadores (máquinas), além de informação de contexto (temperatura, humidade). Baseando-se nos dados obtidos é possível otimizar o processo de manutenção dos equipamentos e controlar melhor todo o processo de

produção.

Por último, é apresentado o processo de rastreamento de encomendas, onde os participantes são os fornecedores e clientes, os serviços de logística e os fabricantes dos veículos de transporte. Para esta situação, os dados gerados são referentes ao produto e materiais a transportar, aos sensores presentes em todo o processo de transporte do pedido. Através destes dados pode-se otimizar o controlo de stocks e de produção, bem como gerir a qualidade dos produtos entregues.

2.4 Resumo do capitulo

Em suma, este capitulo permitiu fazer um levantamento da literatura e descrever os principais protocolos de comunicação, plataformas e arquiteturas IoT. Inicialmente foi recolhida documentação referente aos quatro protocolos principais, tanto em artigos como nas páginas da documentação técnica. A partir da documentação foi efetuada uma análise referente aos quatro protocolos (MQTT, CoAP, AMQP e OPC-UA), que permitiu diferenciá-los relativamente às aplicações de cada um e aos aspetos técnicos (tempo de resposta, *overhead* de pacotes, etc). Com base na análise efetuada, conclui-se que o melhor protocolo, referente ao uso em aplicações industriais, é o OPC-UA, por já ter sido testado com sucesso em vários sistemas industriais, sendo ideal para o controlo destes sistemas. No entanto, protocolos como o MQTT e CoAP são também boas soluções, devido ao grande crescimento que apresentam no mercado das aplicações IoT, sendo que o MQTT é ideal para sistemas de monitorização, devido à utilização de um modelo *publish/subscribe*.

Quanto às plataformas IoT o processo foi idêntico, diferindo apenas nos parâmetros de avaliação. Essa avaliação tem em consideração características como o número de atualizações de que é alvo a base de dados utilizada ou o tipo de protocolos que suporta. Desta comparação resultou que as plataformas IoT mais robustas eram o FIWARE (*Orion* + Agente IoT), o *Microsoft Azure* e o AWS IoT.

Nesta dissertação outro parâmetro a avaliar é a arquitetura adequada a uma sistema industrial complexo. Neste campo foi estudada a arquitetura modelada pelo IDS, resultando desta caracterização aspetos a ter em conta, no desenvolvimento da solução final.

Capítulo 3

Desenvolvimento do CPPS Sniffer

Neste capítulo é descrito o processo de desenvolvimento do *CPPS Sniffer*, bem como as suas funcionalidades principais. O *CPPS Sniffer* consiste no resultado palpável desta dissertação, sendo este desenvolvido com a finalidade de cumprir os objetivos previamente estipulados.

Inicialmente são descritas as tecnologias utilizadas no desenvolvimento do *CPPS Sniffer*. Depois disso, descreve-se com algum grau de detalhe o conceito do *CPPS Sniffer*, nomeadamente a sua arquitetura de rede, as suas principais funcionalidades e a sua estrutura, como aplicação de software distribuída. Por fim, explica-se como funciona e como é constituída a lista de componentes partilhada entre *CPPS Sniffers*.

3.1 Processo de desenvolvimento

O processo de desenvolvimento do *CPPS Sniffer* decorreu de uma forma faseada, através de diferentes etapas. De forma a cumprir os objetivos previamente estabelecidos tais como a aplicação ser completamente distribuída e capaz de efetuar a gestão de dispositivos e outras plataformas, através das respetivas diferentes redes locais.

De seguida foi necessário estabelecer os requisitos que a plataforma *CPPS Sniffer* deveria cobrir, de forma a permitir realizar a ideia estabelecida anteriormente. Consoante os requisito estabelecidos, é necessário definir quais as tecnologias mais adequadas a esta plataforma e averiguar as vantagens e as desvantagens da utilização de uma tecnologia, quando comparada com as restantes. Além de definir os requisitos e tecnologias é importante também estruturar e definir a arquitetura em que a plataforma vai executar. Assim, também é necessário compreender certos conceitos, seja relacionados com a arquitetura ou com a plataforma, de forma a perceber melhor os requisitos e o contexto do *CPPS Sniffer*.

A partir dos requisitos e das tecnologias estabelecidos anteriormente é a altura de modelar a solução pretendida. Nesta primeira fase começa-se por definir a estrutura da solução num nível mais elevado, através de diagramas de pacotes e fluxogramas (diagramas de atividades) e vai-se descendo e modelando cada vez mais a solução, até que se chega um momento em que estão

definidas todas as classes e os respetivos métodos e atributos a desenvolver (diagrama de classes), bem como a interação entre os diferentes objetos (diagrama de sequência).

Quando a estrutura da solução e a iteração entre as componentes estiver terminada transita-se para a fase de desenvolvimento do código em si. Esta fase consiste no desenvolvimento do código e posterior verificação do mesmo. Como auxílio são utilizadas ferramentas que facilitam o desenvolvimento do código como um ambiente de desenvolvimento integrado (IDE), além de ser utilizado um repositório para controlo das versões. De maneira a criar uma cópia de segurança do projeto, bem como fazer uma gestão de versões do projeto desenvolvido, foi utilizado um repositório no *github*, por forma a hospedar este projeto [24]. Para além de ser mantida uma cópia de segurança, a utilização de um repositório é útil, na medida em que permite reverter alterações no código e partilhar o projeto com outros programadores.

3.2 Tecnologias Utilizadas

Antes de descrever o *CPPS Sniffer* e o seu método de funcionamento, neste secção pretende-se identificar e descrever as tecnologias utilizadas para a elaboração e desenvolvimento deste projeto de software. A especificação destas tecnologias baseou-se num trabalho de análise dos requisitos do problema a resolver, de forma a escolher as tecnologias mais indicadas para o desenvolvimento do projeto em questão.

Desta forma de seguida serão descritas as tecnologias usadas, nomeadamente a nível de linguagem de programação, modelo de informação, protocolo utilizado na comunicação entre *CPPS Sniffers* e o repositório utilizado para armazenar o projeto.

3.2.1 Linguagem de programação

Uma das principais tecnologias que foi definida é a linguagem de programação a utilizar. Neste projeto, a linguagem de programação escolhida foi o Java, em detrimento das restantes devido aos seguintes fatores:

- **Independência de plataforma**: A independência de plataforma é resultado da utilização de uma máquina virtual para a interpretação do código (*Java Virtual Machine*), assim para executar um programa desenvolvido em Java, a máquina que vai executar o programa apenas tem de ter instalada a máquina virtual. Desta forma, assegura-se que o programa desenvolvido pode executar em qualquer máquina (Windows, Linux) desde que possua o *Java Runtime Enviroment* (JRE) instalado.

- **Documentação**: A nível de documentação, o Java utiliza *javadocs* para documentar todas as classes, o que auxilia na compreensão do seu funcionamento. Para além dos *javadocs*, encontra-se muita informação *online*, tanto em fóruns (por exemplo *Stack Overflow*) como noutros sites (*Oracle*).

- **Facilidade de utilização**: A linguagem de programação Java, quando associada a um ambiente de desenvolvimento integrado (IDE), como por exemplo o *Netbeans* ou o *Eclipse*, facilita o trabalho do programador, pois possui *plugins* para auto-complementação do código, para compilar o projeto automaticamente, para efetuar *debug* do projeto como também é capaz de indicar automaticamente erros e *warnings*.

- **Desempenho**: Quanto ao desempenho, quando comparada com linguagens de mais baixo nível como C/C++, o Java demonstra um desempenho mais pobre a nível de tempos de execução. Contudo, tendo em conta as restrições temporais deste projeto, estas ficam asseguradas com a utilização de Java como linguagem de programação.

3.2.2 Modelo de informação

O modelo de informação refere-se principalmente ao formato de dados adotado, usado para guardar a lista com todas as componentes presentes na rede. Neste caso, o formato adotado foi o JSON. Este formato é muito utilizado em aplicações Web, devido à sua flexibilidade e facilidade de utilização. A escolha do JSON baseou-se também no facto de já ser utilizada pelo FIWARE na comunicação entre componentes. Como a lista de componentes é bastante volátil, a melhor escolha para o formato do ficheiro foi o JSON, pois advém maior flexibilidade na constante adição de novas componentes.

3.2.3 Protocolo de comunicação

Relativamente aos protocolos de comunicação, o protocolo adotado foi o MQTT. Para um correto funcionamento deste protocolo é necessária a utilização de um *MQTT Broker*. Assim, todos os *CPPS Sniffers* têm de ter um *MQTT Broker* local associado, para facilitar o cumprimento deste requisito, caso o utilizador pretenda, aquando da execução do *CPPS Sniffer*, é iniciado também um *MQTT Broker*.

Este *broker* MQTT consiste numa implementação, em Java, intitulada de *Moquette* [27], que é anexada ao projeto do *CPPS Sniffer*.

3.3 Arquitetura

No panorama industrial atual assume-se que grandes empresas poderão ter várias fábricas espalhas geograficamente pelo mundo, sendo que estas estão conectadas entre si via Internet. Isto já é feito para efeitos de interligação de softwares de gestão de alto nível, de forma às diferentes partes interessadas poderem ter uma visão global sobre a sua produção e tomar decisões estratégicas em relação ao negócio.

Contudo, o problema que está a ser abordado nesta dissertação refere-se à interligação de plataformas e equipamentos de chão de fábrica remotamente, isto é equipamentos e plataformas que estão presentes em diferentes fábricas, assim como representado na Figura - 3.1. Interfaces de comunicação entre equipamentos - M2M, têm vindo a ser exploradas tanto a nível cientifico como

a nível comercial. Contudo, estas soluções estão limitadas à capacidade de interligar equipamentos que estejam presentes na mesma rede local, não considerando a interação entre equipamentos.

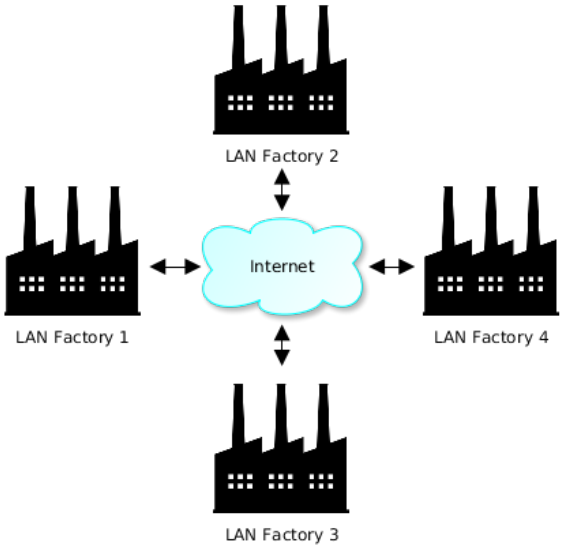

Figura 3.1: Arquitetura completa de rede.

Cada fábrica pode ser constituída por uma ou mais redes locais, sendo que diversos dispositivos e plataformas externas, como *clouds*, podem-se ligar a essa rede, de forma a serem capazes de interagir com outros dispositivos presentes na mesma rede.

Na arquitetura adotada, cada rede local está associada, no mínimo, a um *CPPS Sniffer*, que consiste na componente de rede que faz a descoberta de novos dispositivos dessa rede, de maneira a facilitar a sua interligação, isto é sem necessidade de configurações manuais. O *CPPS Sniffer* pode ser de dois tipos, remoto ou local. *CPPS Sniffers* locais apenas são capazes de operar em redes locais isoladas da Internet, enquanto os *CPPS Sniffers* remotos permitem tanto comunicar com outros *CPPS Sniffers* através da Internet, como também na mesma rede local. Isto significa que com o uso de *CPPS Sniffers* remotos, é possível interligar equipamentos presentes em diferentes redes locais

O *CPPS Sniffer* está encarregue de fazer a gestão do fluxo das publicações efetuadas pelos dispositivos e outros componentes na rede, e de gerirem uma lista que contém todos os componentes presentes da rede. Esta lista funciona como umas páginas amarelas, que devem ser constantemente atualizadas, entre todos os *CPPS Sniffers* de todas as redes. Através desta lista é possível descobrir dispositivos presentes em redes diferentes e abrir canais de comunicação entre estes.

Relativamente aos protocolos suportados e considerando que diferentes dispositivos comunicam usando diferentes protocolos de comunicação, pretende-se que a plataforma suporte o máximo de protocolos possível. Contudo, devido ao extenso trabalho de implementação e normalização de protocolos e modelos de informação dentro da calendarização associada a este projeto, inicialmente o *CPPS Sniffer* foi desenvolvido com suporte apenas ao protocolo MQTT. Isto significa que, nesta fase, dispositivos presentes na rede, que não suportem MQTT, não são capazes de

usar o *CPPS Sniffer* para conetar-se com outros equipamentos ou plataformas. Enquanto que estender o suporte do *CPPS Sniffer* para outros protocolos M2M (CoAP, AMQP) será contemplado como trabalho futuro.

3.3.1 Comunicação via MQTT

Considerando uma visão mais local sobre uma *LAN Factory*, a Figura - 3.2 representa uma dessas redes local (LAN).

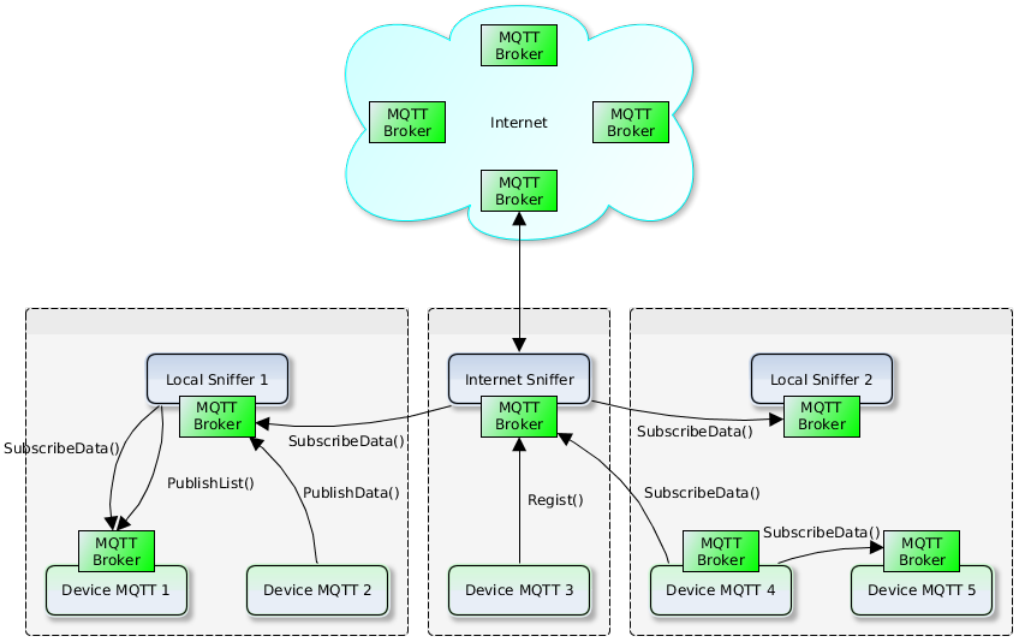

Figura 3.2: Arquitetura da rede com as várias interações via MQTT.

Nesta rede é possível encontrar cinco dispositivos que comunicam através de MQTT, designados de *Device MQTT*, que corresponde a dispositivos físicos que pretendem enviar e/ou receber informação dos restantes dispositivos, usando o protocolo MQTT. Neste caso, é possível criar e associar um *MQTT Broker*, como exemplificado para os *Device MQTT 1*, *Device MQTT 4* e *Device MQTT 5*. Por outro lado, um *MQTT Broker* pode estar associado a outro dispositivo na rede, que neste caso também está associado ao *CPPS Sniffer*. Todos os dispositivos presentes na rede têm de estar associados a um *CPPS Sniffer*. Neste caso, os *Device MQTT 1* e *Device MQTT 2* estão associados ao *CPPS Sniffer Local Sniffer 1*, o *Device MQTT 3* está associado ao *CPPS Sniffer Internet Sniffer* e os *Device MQTT 4* e *Device MQTT 5* estão associados ao *CPPS Sniffer Local Sniffer 2*. Salienta-se que os *Local Sniffer 1* e *Local Sniffer 2* são *CPPS Sniffer* locais, enquanto que o *Internet Sniffer* é um *CPPS Sniffer* remoto.

Antes de começar a enviar os dados recolhidos, inicialmente os *Device MQTT* têm de se registar num *CPPS Sniffer* local ou remoto. Para fazer o registo, o *CPPS Sniffer* ativamente subscreve o tópico de registo no seu próprio *MQTT Broker*, como exemplificado no caso dos *Device MQTT*

2 e *Device MQTT 3*, ou subscreve o tópico de registo no *MQTT Broker* do próprio *Device MQTT*, como exemplificado no caso dos *Device MQTT 1*, *Device MQTT 4* e *Device MQTT 5* .

A partir do momento que o *Device MQTT* está registado, este está em condições de publicar os dados que vai obtendo, seja no seu *MQTT Broker* ou no *MQTT Broker* do *CPPS Sniffer* respetivo. Os *CPPS Sniffers* locais subscrevem os tópicos que são usados pelos *Device MQTT* para publicar informação. Por outro lado, os *CPPS Sniffers* remotos subscrevem toda a informação subscrita por todos os *CPPS Sniffers* locais, de forma a difundir a informação dos tópicos subscritos a *Device MQTT* presentes noutras *LAN Factories*, usando nomeadamente os seus *CPPS Sniffers* remotos, que tenham subscrito os mesmos tópicos.

Para facilitar o registo de novos *CPPS Sniffers*, é constantemente publicada em todos os *MQTT Brokers* da rede uma lista atualizada de todos os *CPPS Sniffers* e dispositivos presentes na rede global. Esta lista permite aos *CPPS Sniffers* e aos *Device MQTT* saber a localização na rede, através do endereço IP e porta, de todas as componentes que se encontram na rede. Desta forma, o uso da lista simplifica a operação de subscrição de tópicos por parte dos *Device MQTT*, como exemplificado nas operações entre o *CPPS Sniffer* remoto e o *Device MQTT 4*.

Como foi referido anteriormente o *Device MQTT* pode ou não optar por criar o seu próprio *MQTT Broker* local, o que pode ser uma vantagem ou desvantagem, dependendo esta opção dos seguintes aspetos:

- **Sobrecarga do *MQTT Broker***: Caso todos os *Device MQTT* utilizassem o *MQTT Broker* do *CPPS Sniffer*, este seria facilmente sobrecarregado. Para evitar esta situação os *Device MQTT* que geram mais dados devem utilizar o seu próprio *MQTT Broker*, desta forma é efetuada uma interpolação dos dados antes de serem publicados no *MQTT Broker* do *CPPS Sniffer* respetivo.

- **Comunicação *M2M***: A comunicação M2M constitui uma vantagem e consiste em os *Devices MQTT* poderem comunicar diretamente entre si (M2M), o que é exemplificado entre os *Device MQTT 4* e *Device MQTT 5*. Este tipo de comunicação é mais vantajoso pois diminui-se o tempo de entrega das mensagens, devido a não ser necessária a intervenção de um intermediário (*CPPS Sniffer*).

3.4 Definição do *CPPS Sniffer*

O *CPPS Sniffer* é a componente que deve estar presente em todas as redes locais, de forma a fazer a gestão de dispositivos MQTT presentes nestas. O *CPPS Sniffer* facilita a inclusão de novos componentes na rede, sejam eles dispositivos MQTT presentes no chão de fábrica, como sensores, equipamentos industriais, *Automated Guided Vehicles* (AGVs) e manipuladores robóticos, ou mesmo plataformas externas, prestadoras de serviços sobre os dados gerados no chão de fábrica, como *clouds* ou MES. Desta forma, e baseado nos requisitos definidos, o *CPPS Sniffer* apresenta os seguintes requisitos:

- **Descoberta de dispositivos**: Sempre que um novo dispositivo se liga a uma rede local, o *CPPS Sniffer* presente na rede deve ser capaz de detetá-lo, sem ser necessário configurar manualmente os IPs de cada um dos componentes. Para isto, são utilizados endereços *multicast*, que permitem tanto aos novos dispositivos como aos novos *CPPS Sniffers*, descobrirem se já existe algum *CPPS Sniffer* na rede.

- **Registo dos dispositivos**: Antes de começarem a produzir e enviar dados, os dispositivos devem efetuar um registo, que consiste no envio de um ficheiro no formato JSON para o *CPPS Sniffer* correspondente, com todos os parâmetros que caracterizam o dispositivo. Ao receber este ficheiro, o *CPPS Sniffer* deve adicioná-lo à lista dos dispositivos.

- **Pré-processamento dos dados**: Os dispositivos presentes na rede local produzem e atualizam constantemente um tópico com novos dados, que é subscrito pelo *CPPS Sniffer*. Caso este reencaminhasse os dados automaticamente para o resto da rede, causar-se-ia uma sobrecarga nos restantes *MQTT Brokers*. Para evitar esta situação, os dados antes de serem enviados têm de ser pré processados, que consiste maioritariamente numa interpolação de dados.

- **Gestão da lista dos dispositivos**: Quando um novo dispositivo efetua o registo, são adicionados a uma lista os parâmetros que o caracterizam, entre eles o IP local, o identificador do dispositivo e os tópicos em que publica dados. Esta lista deve ser constantemente atualizada entre os *CPPS Sniffers*, para existir transparência entre todas as redes locais, ou seja todos os *CPPS Sniffers* têm conhecimento de todos os dispositivos presentes em todas as redes.

- **Disponibilizar dados a dispositivos de outras redes**: Para tornar a rede o mais distribuída possível, o *CPPS Sniffer* deve ser capaz de fornecer dados dos dispositivos da sua rede local a dispositivos externos. Esta funcionalidade permite, em caso de falha de um *CPPS Sniffer*, que o resto do sistema não seja afetado por essa falha, conseguindo funcionar normalmente.

3.4.1 Tipos de *CPPS Sniffer*

Como foi referido anteriormente 3.4 o *CPPS Sniffer* pode ser configurado de duas formas, nomeadamente como local ou como remoto. Em termos de desenvolvimento, o *CPPS Sniffer* remoto constituiu uma versão do *CPPS Sniffer* local com maior número de funcionalidades.

- ***CPPS Sniffer* local**: Relativamente ao *CPPS Sniffer* local, este deve ser capaz de efetuar todas as funcionalidades descritas anteriormente 3.4, numa rede local.

- ***CPPS Sniffer* remoto**: Quanto ao *CPPS Sniffer* remoto, este deve ter a capacidade de efetuar todas as funcionalidades de um *CPPS Sniffer* local. Para além disso, deve ser capaz de reencaminhar o tráfego de dados dos *CPPS Sniffers* locais associados através da Internet, de maneira a permitir que dispositivos presentes noutras redes consigam conectar-se remotamente.

3.5 Lista das entidades

Esta lista consiste num ficheiro no formato JSON, onde são guardados todos os *CPPS Sniffers*. Para cada *CPPS Sniffer*, são armazenados os parâmetros que o caracterizam, que são indispensáveis para manter todos os componentes da rede permanentemente conectados. A estrutura para a lista contempla uma série de parâmetros de caracterização, nomeadamente:

- **Identificador**: Identificador único entre os *CPPS Sniffers* que permite diferenciá-los.
- **Tipo**: O *CPPS Sniffer* pode ser de 2 tipos, remoto ou local. Caso seja local, apenas é necessário especificar os parâmetros do *MQTT Broker* local. Por outro lado, caso seja do tipo remoto, é necessário adicionar os parâmetros do *MQTT Broker* remoto.
- ***MQTT Broker* remoto**: *MQTT Broker* que executa numa máquina com ligações remotas, via Internet, e permite efetuar a comunicação com o resto dos *CPPS Sniffers*.
  - **IP público**: Endereço público ou domínio ao qual está associado o respetivo *MQTT Broker*.
  - **Porta**: Porta utilizada pelo *MQTT Broker* local. Por norma, é adotada a porta 1883, para o protocolo MQTT.
  - **Nome de utilizador**: Para aumentar a segurança neste *MQTT Broker*, é necessário efetuar uma autenticação para publicar ou subscrever informação.
  - **Palavra-passe**: Chave de segurança associado ao nome de utilizador previamente descrito.
- ***MQTT Broker* local**: *MQTT Broker* que executa na mesma máquina onde está a correr o *CPPS Sniffer*.
  - **IP local**: Endereço privado ao qual está associado este *MQTT Broker*.
  - **Porta**: Porta utilizada pelo *MQTT Broker* local. Por norma é adotada a porta 1883, para o protocolo MQTT.
- **Rede Local**: De forma a agrupar os *CPPS Sniffers*, é utilizado um endereço *multicast* associado a uma rede. Assim, cada *CPPS Sniffer* pertencente a esta rede deve possuir os mesmo parâmetros de rede local.
  - **Identificador**: Identificador que permite associar diferentes *CPPS Sniffers*, com base na rede local a que pertencem, e identificar as diferentes redes locais.
  - **IP *multicast***: Endereço *multicast* (intervalo que vai do 224.0.0.0 até 239.255.255.255) ao qual está uma rede de um ou vários *CPPS Sniffers*.
- **Protocolo**: O protocolo suportado pelo *CPPS Sniffer* é também registado e permite saber se existe compatibilidade entre as componentes.

- **Dispositivos**: Todos os dispositivos presentes na rede local, associados ao *CPPS Sniffer* também são registados.
  - **Identificador**: Identificador único entre os dispositivos do respetivo *CPPS Sniffer*.
  - **Tipo**: O dispositivo pode ser de 2 tipos, nomeadamente com ou sem *MQTT Broker* local. Caso tenha *MQTT Broker* local, devem ser especificados os seus parâmetros. Caso contrário, devem ser adotados os parâmetros do *MQTT Broker* local do respetivo *CPPS Sniffer*.
  - ***MQTT Broker* local**: *MQTT Broker* associado ao dispositivo ou ao respetivo *CPPS Sniffer*.
    - **IP local**: Endereço privado ao qual está associado o respetivo *MQTT Broker*.
    - **Porta**: Porta utilizada pelo *MQTT Broker* local. Por norma é adotada a porta 1883, para o protocolo MQTT.
  - **Protocolo**: Protocolo suportado pelo dispositivo.
  - **Atributos**: Lista de parâmetros onde são apresentadas as características do dispositivo, como por exemplo o tipo de sensor (temperatura, tensão, pressão). Para cada atributo, deve existir na lista um identificador e o tipo de variável utilizado nos valores produzidos (inteiro, texto).

A lista funciona como um recurso partilhado, numa filosofia de descentralização, ou seja cada *CPPS Sniffer* mantém uma cópia constantemente atualizada da lista. Uma das vantagens desta abordagem é a tolerância a falhas, ou seja em caso de falha de um *CPPS Sniffer*, esta não tem um impacto significativo no funcionamento das redes a nível global, sendo que o resto de redes locais funcionaria normalmente. Outra vantagem é o fácil acréscimo de novas componentes à rede, podendo ser estas novas *clouds* ou novos dispositivos, desde que suportem comunicação via MQTT.

3.6 Estrutura da solução

De forma a apresentar a arquitetura do *CPPS Sniffer*, identificam-se e descrevem-se os diferentes módulos da solução desenvolvida, estando estes apresentados na Figura - 3.3.

Os pacotes/módulos representados no diagrama estão organizados de forma a tornar a análise da estrutura do *CPPS Sniffer* mais fácil e intuitiva. As classes responsáveis pela inicialização do *CPPS Sniffer* estão presentes no pacote *init system*, sendo que estas classes permitem o registo do *CPPS Sniffer* na rede, a subscrição da lista de componentes, a iniciação de *threads* associadas aos pacotes *sniffer* e *discovery*.

Relativamente ao pacote *discovery*, este contém associadas as classes que permitem fazer a descoberta de novos *CPPS Sniffers* e/ou dispositivos. Quanto ao pacote *sniffer*, este contém classes que permitem subscrever tanto o *MQTT Broker* local como o remoto, além de incluir também as classes responsáveis pela implementação do protocolo para a comunicação entre *CPPS Sniffers*.

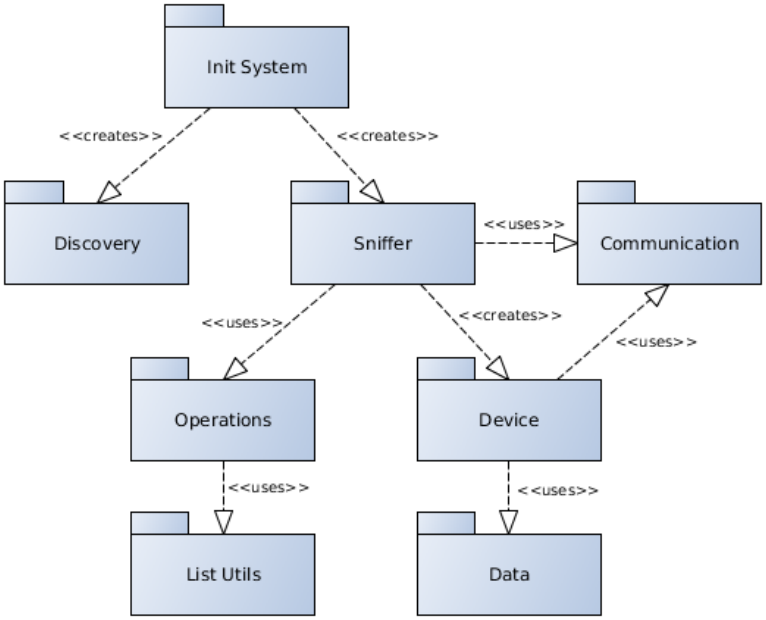

Figura 3.3: Diagrama de pacotes.

A comunicação via MQTT é efetuada através das classes associadas ao pacote *communication*, enquanto que os pedidos/mensagens recebidos pelo *MQTT Broker* associado ao *CPPS Sniffer*, são processados por classes presentes no pacote *operations*.

O pacote *list utils* agrupa classes que habilitam a leitura ou a escrita no ficheiro JSON responsável por guardar as componentes presentes na rede. Relativamente, ao tratamento dos dados provenientes dos dispositivos, este é efetuado pelas classes presentes no pacote *data*, enquanto que o pacote *device*, contém as classes responsáveis pela subscrição dos *MQTT Brokers* associados aos dispositivos.

3.7 Inicialização do *CPPS Sniffer*

Inicialmente o *CPPS Sniffer* deve passar por uma série de etapas, até poder efetuar o normal processamento. Estas etapas estão representadas na Figura - 3.4.

O processo de inicialização do *CPPS Sniffer* arranca com a leitura de um ficheiro de configuração (descrição disponível em 3.7.1. A partir da leitura deste ficheiro de configurações, o programa é capaz de inferir se é necessário iniciar um *MQTT Broker* local, e se o *CPPS Sniffer* é remoto ou local. Após verificar se é necessário iniciar um *MQTT Broker* local (o utilizador pode optar por iniciar automaticamente o *MQTT Broker* ou por passar os parâmetros de outro *MQTT Broker*), o *CPPS Sniffer* verifica se existem outros *CPPS Sniffers* na rede. Se essa condição se verificar, é subscrito nesse *CPPS Sniffer* o tópico responsável pela publicação da lista. Em caso contrário, é verificado se o *CPPS Sniffer* é remoto ou local, sendo que se o *CPPS Sniffer* for remoto, a lista é

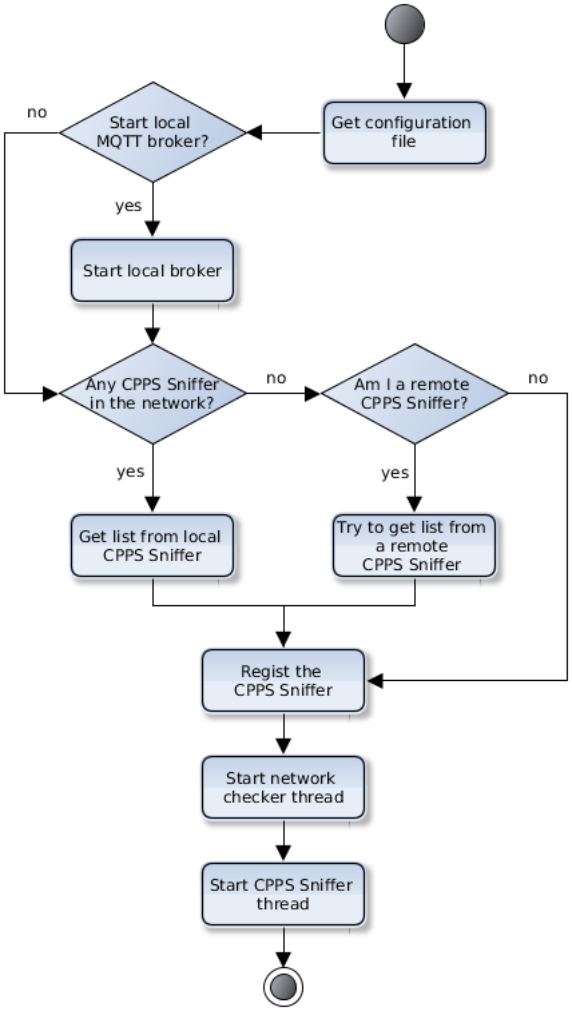

Figura 3.4: Diagrama de atividade da inicialização de um *CPPS Sniffer*.

obtida via Internet, através de outro *CPPS Sniffer* remoto, caso não exista nenhum *CPPS Sniffer* é iniciada uma nova lista.

Para finalizar este processo, o *Digital Twin* do *CPPS Sniffer* é adicionado à lista local e é enviada uma mensagem para os restantes *CPPS Sniffers* atualizarem a sua versão da lista. Posteriormente são iniciadas as *threads* responsáveis por verificar a existência de novos *CPPS Sniffers*/dispositivos na rede e pelo funcionamento do *CPPS Sniffer*.

3.7.1 Configuração do *CPPS Sniffer*

De maneira a facilitar a utilização do *CPPS Sniffer*, foi criado um ficheiro de configuração, onde são especificados vários parâmetros necessários para o normal funcionamento deste componente. Entre estes parâmetros, pode-se encontrar um identificador do *CPPS Sniffer*, que deve ser único em toda a rede e o tipo de *CPPS Sniffer* (local ou remoto). Relativamente ao *MQTT Broker* local, é possível parametrizar se é necessário iniciar o *MQTT Broker* ou não, e qual é o IP a utilizar

no *MQTT Broker*. Para além disso, o IP pode ser detetado automaticamente pelo *CPPS Sniffer* ou especificado pelo utilizador, caso este assim pretenda.

Caso o *CPPS Sniffer* seja do tipo remoto, é necessário especificar parâmetros relativos ao *MQTT Broker* remoto, como o IP, porta, nome de utilizador e palavra-passe. Na eventualidade de se pretender conectar a uma rede remota (por exemplo outra fábrica), é necessário caracterizar os parâmetros associados ao *MQTT Broker* remoto responsável por essa rede, de forma a obter a lista de componentes.

A utilização de endereços *multicast* permite configurar diferentes redes dentro de uma rede local. Contudo é também necessário caracterizar certos parâmetros destas redes, como o endereço *multicast* utilizado, o identificador da rede *multicast* e a porta utilizada.

3.7.2 *Thread* associada ao *CPPS Sniffer*

A *thread* responsável pelo processamento dos *MQTT Brokers* associados ao *CPPS Sniffer* está representada na Figura - 3.5.

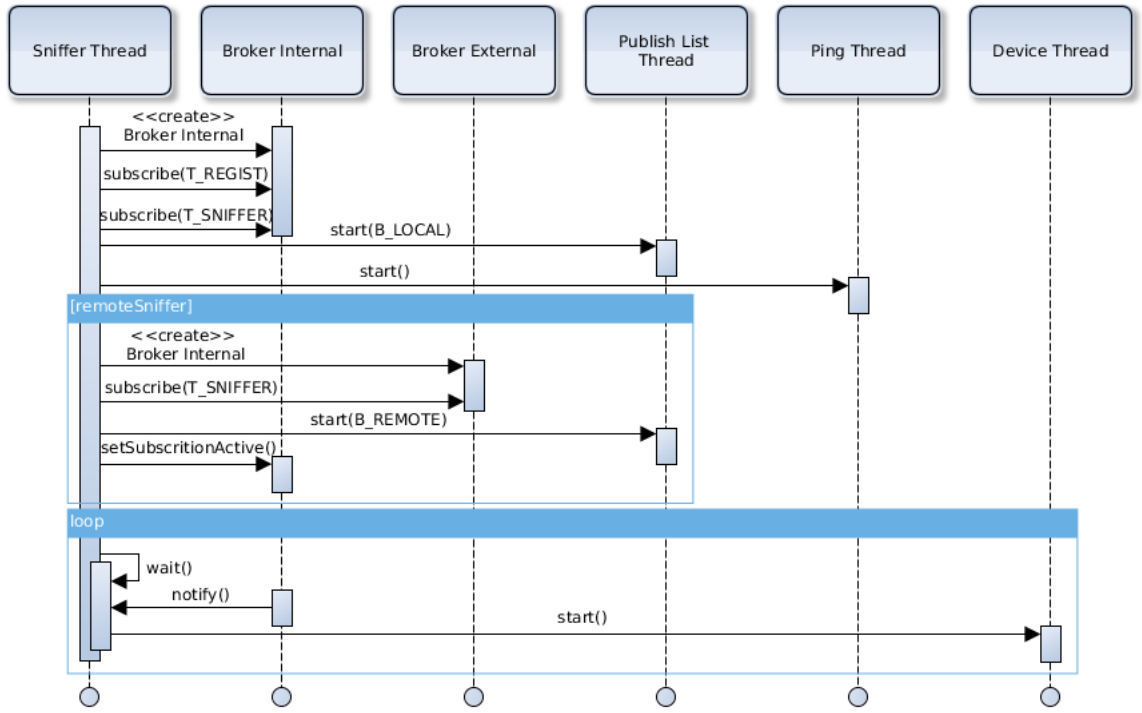

Figura 3.5: Diagrama de sequência da *thread* associada ao *CPPS Sniffer*.

O processo representado neste diagrama de sequência inicia-se com a instanciação de um objeto da classe *broker internal*. Este objeto é capaz de subscrever tópicos MQTT associados ao *MQTT Broker* local e posteriormente processar as mensagens recebidas nesses tópicos. Este objeto subscreve dois tópicos, um associado ao registo de dispositivos e outro responsável pela comunicação com outros *CPPS Sniffers*. De seguida, são iniciadas duas *threads* independentes, uma responsável pela publicação da lista de componentes no *MQTT Broker* local e outra encarregue

de enviar uma mensagem de *ping* a outros *CPPS Sniffers*, com o intuito de indicar que este *CPPS Sniffer* está a funcionar corretamente.

Após a criação destas *threads* e caso se trate de um *CPPS Sniffer* remoto é instanciado um objeto da classe *broker external*. Este objeto é semelhante ao associado à classe *broker internal*, tendo como diferença o facto de este se conectar a um *MQTT Broker* externo. Relativamente aos tópicos subscritos, neste caso apenas é necessário subscrever o tópico no qual são trocadas mensagens entre *CPPS Sniffers*. Tal como no *MQTT Broker* local, no caso do *MQTT Broker* remoto, é necessário também iniciar uma *thread* encarregue de publicar a lista de componentes neste *MQTT Broker*. O método *setSubscritionActive()* permite ativar a subscrição dos dados publicados no *MQTT Broker* local pelos dispositivos. Esta subscrição permite que os dados produzidos pelos dispositivos sejam reencaminhados pelo *CPPS Sniffer* remoto para a Internet.

A *thread* associada ao *CPPS Sniffer* nunca termina o seu processamento, pois entra num ciclo onde fica permanentemente à espera de receber uma notificação de que se registou um novo dispositivo. A partir desse momento, é iniciada uma nova *thread* que se conecta ao *MQTT Broker* do novo dispositivo e subscreve os tópicos associados a este.

3.8 Comunicação

Como foi referido na Secção 2.1.1, o protocolo MQTT utiliza 2 campos cruciais nas suas mensagens, nomeadamente o tópico que permite agrupar as mensagens de acordo com um tema e os dados que podem conter variáveis de qualquer tipo. Desta forma, o *CPPS Sniffer* utiliza diferentes tópicos consoante o tema das mensagens. Esses tópicos são:

- **`/getlist`**: É no tópico *getlist* que a lista com todos os *CPPS Sniffers* e dispositivos vai sendo publicada. Este tópico é constantemente atualizado pelos *CPPS Sniffers* nos seus *MQTT Brokers* e nos *MQTT Brokers* dos seus dispositivos. Assim quando um novo *CPPS Sniffer* ou dispositivo se conectar à rede, ele deve subscrever este tópico para ter conhecimento das componentes já existentes na rede.

- **`/registdevice`**: O registo dos dispositivos é efetuado no tópico *registdevice*. Neste tópico é publicado um ficheiro JSON por parte do dispositivo, sendo utilizado pelo *CPPS Sniffer* correspondente ao *MQTT Broker*, para adicionar um novo dispositivo à lista.

- **`/device_id/attrs/object_id`**: Este tópico está representado de uma forma genérica, pois para cada atributo de cada dispositivo, estará associado um destes tópicos. São nestes tópicos onde são publicados os dados capturados por cada dispositivo. Desta forma, o parâmetro *device_id* deve ser substituído pelo identificador com que o dispositivo se registou no *CPPS Sniffer* e o *object_id* corresponde ao identificador do atributo.

- **`/sniffercommunication`**: Para manter a lista constantemente atualizada entre *CPPS Sniffers*, é necessário que estes partilhem informação entre eles tanto no registo de novos *CPPS Sniffers*/dispositivos, como também na remoção destes. Para além disto, é necessário

que os *CPPS Sniffers* enviem entre eles mensagens de *alive*, como forma de os restantes saberem quando ocorrem falhas num *CPPS Sniffer*. Desta forma, é utilizado o tópico *sniffercommunication* para a troca de mensagens entre os *CPPS Sniffers*.

3.8.1 Métodos utilizados

Para efetuar a comunicação entre *CPPS Sniffers* ou com dispositivos foram implementados diversos métodos que permitem efetuar a comunicação de diversas formas. Na Figura - 3.6 está apresentado o diagrama de classes do pacote responsável pela comunicação via MQTT. Este pacote está implementado utilizando uma biblioteca de comunicação MQTT desenvolvida no âmbito do projeto *Eclipse Paho* [7].

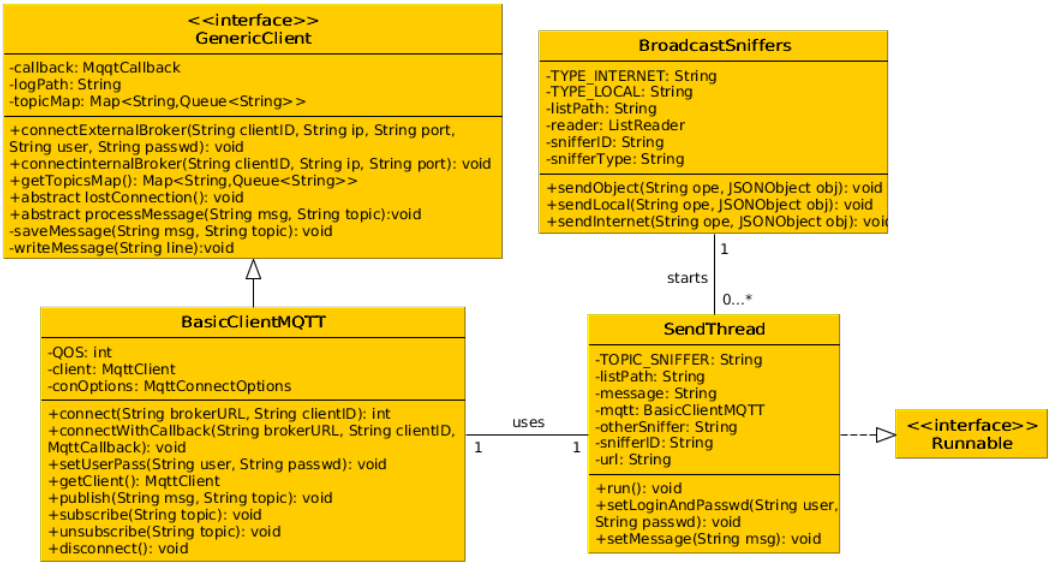

Figura 3.6: Diagrama de classes associado ao pacote responsável pela comunicação.

Em termos de implementação inicialmente foi desenvolvida uma classe que permite efetuar operações de comunicação básicas (classe *BasicClientMQTT*), como conectar-se a um *MQTT Broker*, efetuar publicações e subscrever tópicos. A conexão através desta classe pode ser efetuada ou não com *callback*, sendo que este *callback* é utilizado sempre que é necessário receber mensagens, pelo que só faz sentido utilizar a conexão com *callback* sempre que se subscrever a algum tópico. Deste modo, na *interface GenericClient* é implementado o *callback* e os respetivos métodos abstratos, responsáveis por receber novas mensagens e verificar se existe conexão com o *MQTT Broker*. Esta classe herda todos os métodos da classe *BasicClientMQTT* e é utilizada sempre que é necessário subscrever tópicos a partir de *MQTT Brokers*. Ela permite efetuar conexão a *MQTT Brokers* com ou sem autenticação, além de guardar um registo de todas as mensagens recebidas pelo *CPPS Sniffer* correspondente.

Quando é apenas necessário publicar dados num *MQTT Broker* é utilizada a classe *SendThread*, que implementa uma *thread* responsável por publicar uma mensagem nesse *MQTT Broker*.

Esta *thread* é utilizada pela classe *BroadcastSniffers*, que permite enviar para todos os *CPPS Sniffers* remotos ou na mesma rede uma determinada mensagem, dependendo se o emissor se trata de um *CPPS Sniffer* remoto ou local.

3.9 Interação com dispositivos

A interação entre *CPPS Sniffer* e dispositivo foi projetada de maneira a ser o mais simples possível. Desta forma, os diferentes modos de interação entre o *CPPS Sniffer* e o dispositivo, podem ser modelados pelos casos de utilização, como representado na Figura - 3.7.

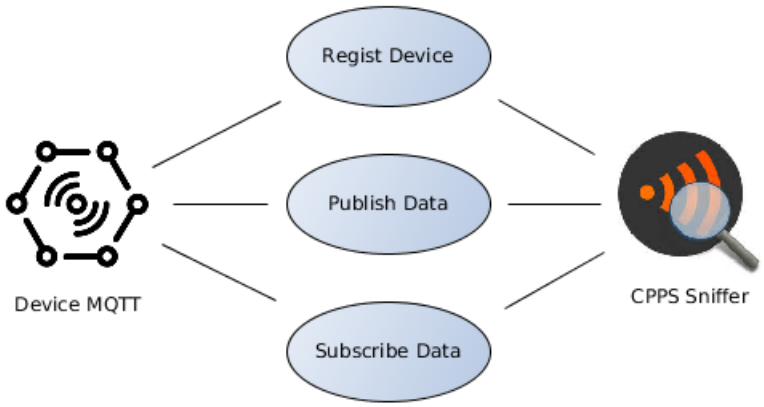

Figura 3.7: Diagrama de casos de utilização entre o *CPPS Sniffer* e o dispositivo.

No diagrama de casos de utilização estão representados os três tipos de interação entre os dispositivos e o *CPPS Sniffer*. Antes de iniciar a publicação/subscrição de dados é obrigatório efetuar o registo do dispositivo, para o *CPPS Sniffer* saber quais os atributos que caracterizam o dispositivo. De seguida o dispositivo está em condições de efetuar publicações nos tópicos registados e subscrever os tópicos que pretender.

Desta forma, o registo do dispositivo pode ser efetuado de diferentes formas, dependendo se o dispositivo possuir o IP do *CPPS Sniffer* ou não. Caso não esteja configurado o IP do *CPPS Sniffer* no dispositivo este pode optar por enviar um pacote especifico para um endereço *multicast* de forma a ser descoberto. A partir desse momento, o dispositivo passa a efetuar todas as suas publicações no próprio *MQTT Broker* (publicação de registo incluída). Na eventualidade do dispositivo ter configurado o IP do *MQTT Broker*, este pode optar por inicialmente publicar a mensagem de registo no *MQTT Broker* do *CPPS Sniffer*, e a partir desse momento, efetuar todas as outras publicações no seu próprio *MQTT Broker*. Caso o dispositivo não possua *MQTT Broker*, este apenas necessita de efetuar todas as suas publicações no *MQTT Broker* do *CPPS Sniffer*.

3.9.1 Tipos de dispositivos

Como foi referido anteriormente, os dispositivos suportados pelo *CPPS Sniffer* podem ser de 2 tipos, com ou sem *MQTT Broker*. A utilização ou não de um *MQTT Broker* condiciona o

funcionamento do *CPPS Sniffer*.

- **Dispositivo com *MQTT Broker***: Quando um novo dispositivo usa um *MQTT Broker* diferente do utilizado pelo *CPPS Sniffer*, é necessário iniciar uma *thread* responsável pela subscrição dos tópicos associados ao novo *MQTT Broker*, assim como representado no diagrama de sequência da Figura - 3.5, quando é instanciado um objeto da classe *device thread*. A partir desse momento é necessário efetuar o reencaminhamento dos dados publicados no *MQTT Broker* do dispositivo para o *MQTT Broker* do *CPPS Sniffer*.

- **Dispositivo sem *MQTT Broker***: No caso do dispositivo não possuir *MQTT Broker* é muito mais simples, pois não é necessário reencaminhar os dados do dispositivo. Neste caso, apenas é necessário efetuar o registo do dispositivo na rede.

3.9.2 Gestão de dados dos dispositivos

A gestão dos dados provenientes dos dispositivos é efetuada tanto pelo *CPPS Sniffer* local como pelo remoto. Os dados são publicados em *MQTT Brokers*, que posteriormente reencaminham estes dados para os respetivos subscritores. Nesta arquitetura são utilizados tanto *MQTT Brokers* remotos (possuem um IP público), como *MQTT Brokers* locais (possuem um IP local). Desta forma, os dados são produzidos inicialmente em *MQTT Brokers* locais e posteriormente são reencaminhados para o *MQTT Broker* remoto associado.

Assim, o *CPPS Sniffer* local tem de garantir que todos os dados produzidos pelos seus dispositivos são reencaminhados para o seu *MQTT Broker*, no caso dos dispositivos possuírem *MQTT Brokers* locais. Nesse caso, pode ser efetuada uma interpolação dos dados provenientes do *MQTT Broker* do dispositivo, antes de reencaminha-los. Esta interpolação consiste numa média dos últimos n valores armazenados pelo *CPPS Sniffer*.

Garantindo que todos os dados estão a ser reencaminhados para os *MQTT Brokers* associados aos *CPPS Sniffers* locais, o *CPPS Sniffer* remoto apenas tem de assegurar que subscreve os dados publicados nos *CPPS Sniffers* locais, de maneira a reencaminha-los para o *MQTT Broker* remoto. Assim, os dados de todos os dispositivos presentes nessa rede local, podem ser acedidos por uma máquina com ligação à Internet e com a respetiva autorização (autenticação necessária). Desta forma, é garantida a transparência entre todas as redes locais.

Quanto à subscrição de dados, o dispositivo que pretenda efetuá-la inicialmente terá de subscrever a lista de componentes, de maneira a saber o IP do *MQTT Broker*, onde estão a ser publicados estes dados. É de salientar que caso os tópicos subscritos não pertençam a um *MQTT Broker* da mesma rede local, é necessário efetuar essa subscrição ao nível do *MQTT Broker* remoto.

3.10 Interação entre *CPPS Sniffers*

Como foi referido anteriormente, as mensagens trocadas entre *CPPS Sniffers* estão associadas ao tópico *sniffercomunication*. Assim, as mensagens enviadas com este tópico associado contêm

um campo no inicio da mensagem onde é indicada a operação a efetuar no *CPPS Sniffer* que recebeu a mensagem. O *CPPS Sniffer* suporta 4 tipos de operações, nomeadamente o registo de novas componentes, a remoção de *CPPS Sniffers* ou dispositivos e a verificação da conexão entre *CPPS Sniffers* (*ping*). Estas 4 operações estão representadas no diagrama de atividades da Figura - 3.8.

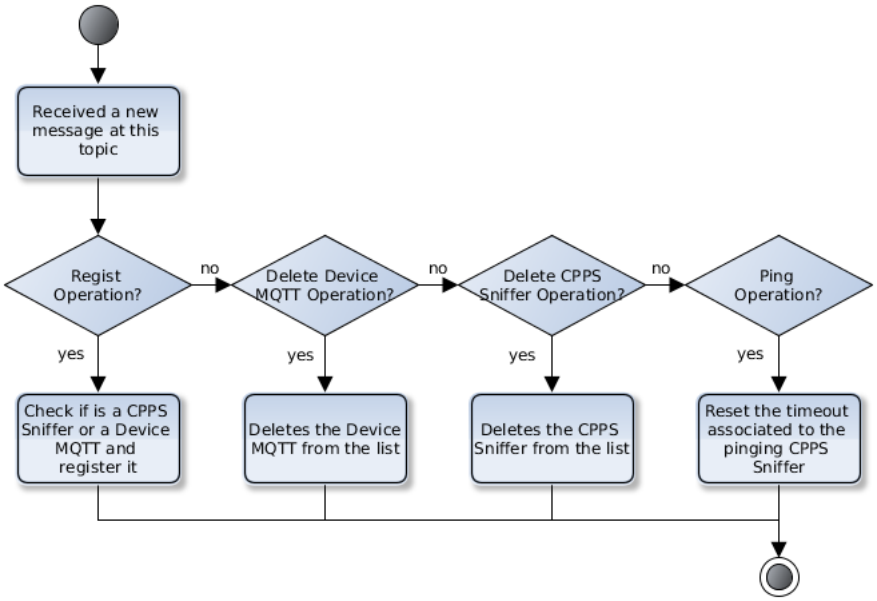

Figura 3.8: Diagrama de atividade da receção de uma nova mensagem no tópico *sniffercommunication*.

3.10.1 Registo de novos *CPPS Sniffers*/dispositivos

A operação *regist* permite o registo de novos *CPPS Sniffers*/dispositivos por parte do *CPPS Sniffer* que recebeu a mensagem. A sequência de processos efetuados pelo *CPPS Sniffer* está descrita na Figura - 3.9.

Inicialmente é verificado se o *CPPS Sniffer* que recebeu a mensagem é local ou remoto. Caso seja remoto e a mensagem proceda da Internet, a mensagem é reencaminhada para os *CPPS Sniffers* locais pertencentes à mesma rede. Por outro lado, caso proceda da rede local, é reencaminhada para todos os *CPPS Sniffers* remotos com ligação à Internet. Este processamento ocorre através do método *middleComunication()*, que se baseia num método que faz *broadcast* para um certo tipo de *CPPS Sniffers*.

Posteriormente é adicionada uma nova componente à lista, através de uma instância da classe *add things*. Neste caso, é verificado se é um novo *CPPS Sniffer* ou se já existe o novo dispositivo associado a este *CPPS Sniffer*. A adição do novo elemento à lista ocorre através do método *addSniffer()*, caso seja um novo *CPPS Sniffer*, ou através do método *addDevice()*, caso se trate de um novo dispositivo.

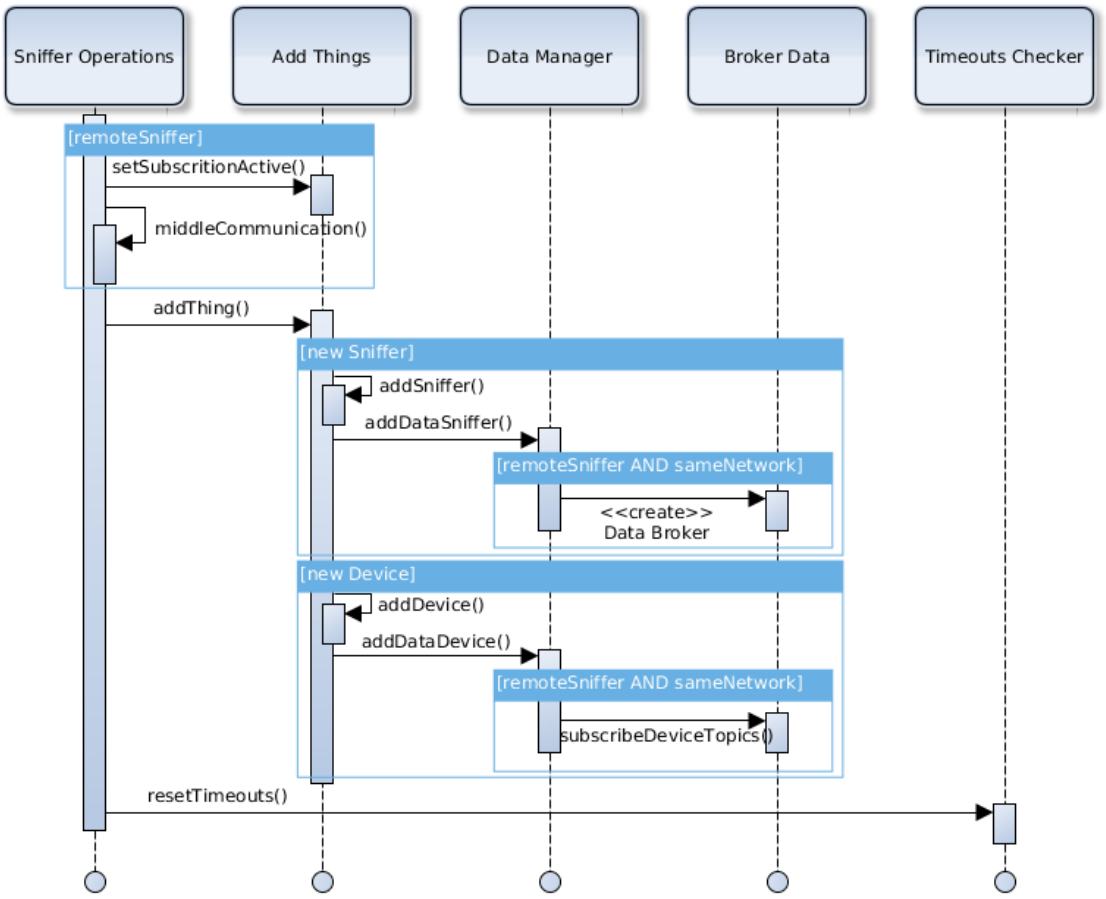

Figura 3.9: Diagrama de sequência de registo de um *CPPS Sniffer*/dispositivo.

Caso o *CPPS Sniffer* que recebeu o pedido seja remoto e o *CPPS Sniffer* registado pertença à mesma rede local é instanciado um objeto (*broker data*) responsável por subscrever todos os dados associados aos dispositivos deste *CPPS Sniffer* e posteriormente é adicionado ao mapa onde se encontram todos os objetos do tipo *broker data*. Assim, sempre que se registar um dispositivo associado a este *CPPS Sniffer* serão subscritos pelo *CPPS Sniffer* remoto todos os tópicos associados ao dispositivos (método *subscribeDeviceTopics()*).

Como foi adicionado um *CPPS Sniffer*/dispositivo é necessário atualizar o mapa encarregue de verificar os *timeouts* entre *CPPS Sniffers*, sendo este realizado pela execução do método *resetTimeouts()*.

3.10.2 Remoção de *CPPS Sniffers*/dispositivos

Os processos de remoção de *CPPS Sniffers* ou de dispositivos são bastante semelhantes entre si. Relativamente à remoção de dispositivos, o processo responsável está ilustrado na Figura - 3.10.

O processo de remoção de um dispositivo desencadeia-se com a receção de uma mensagem com o campo tipo de operação preenchido com o valor *deletedevice*. A partir desse momento, caso

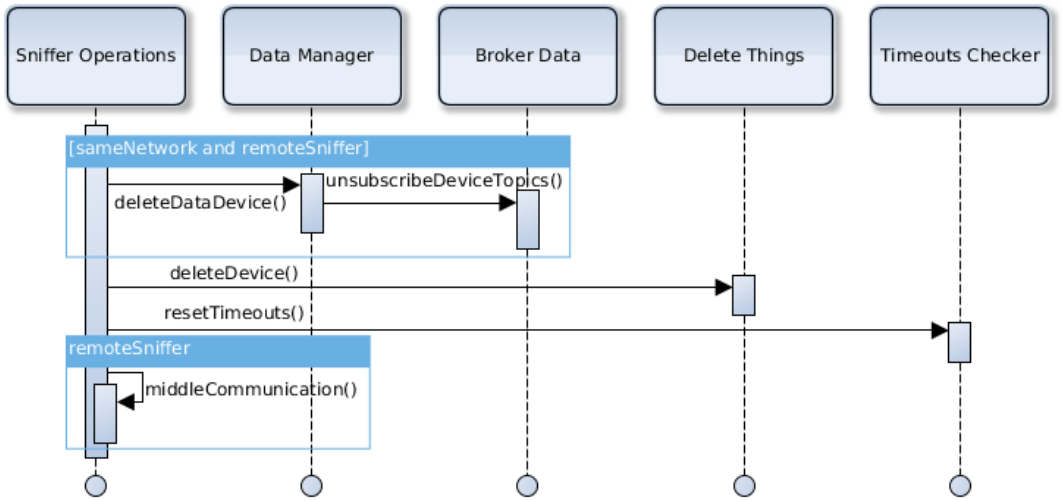

Figura 3.10: Diagrama de sequência da remoção de um dispositivo.

o *CPPS Sniffer* que recebeu o pedido seja remoto e ambos os *CPPS Sniffers* pertençam à mesma rede local, é executado o método *unsubscribeDeviceTopics()*, que remove os tópicos nos quais o dispositivo produz dados e remove o objeto *broker data* associado ao *CPPS Sniffer* responsável por este dispositivo, caso não se verifique esta condição estes métodos são ignorados e prossegue-se para o próxima instrução. Posteriormente é removido da lista o dispositivo, através do método *deleteDevice()* e é efetuado um *reset* no mapa de *timeouts*, tal como descrito na Secção 3.10.1. No final, é executado o método *middleComunication()*, caso o *CPPS Sniffer* que recebeu o pedido seja remoto tal como acontece na operação de registo 3.10.1.

Quanto ao processo de remoção de um *CPPS Sniffer*, este é semelhante ao processo de remoção de um dispositivo, com a diferença que é apagado o objeto *broker data* associado ao *CPPS Sniffer* a remover, ao invés de remover a subscrição dos tópicos, como acontece quando se remove um dispositivo.

3.11 Mecanismos de deteção de novas componentes na rede

Para detetar novas componentes na rede são utilizados endereços *multicast*. Na comunicação *multicast* inicialmente diferentes máquinas conectam-se ao mesmo endereço *multicast*. Quando uma nova máquina envia um pacote para este endereço todas as máquinas que estão conectadas à rede recebem este pacote.

A utilização desta metodologia permite criar diferentes sub-redes dentro de uma rede (associando um endereço *multicast* a cada rede), como também é possível utilizar o mesmo endereço para a rede inteira. No diagrama de atividades presente na Figura - 3.11 está representado todo o processo de deteção, tanto de novos *CPPS Sniffers* como dispositivos.

O processo inicia-se com a criação de um *socket multicast*, que se junta ao grupo associado a um certo endereço *multicast* previamente estabelecido. A partir deste momento, a *thread* fica à

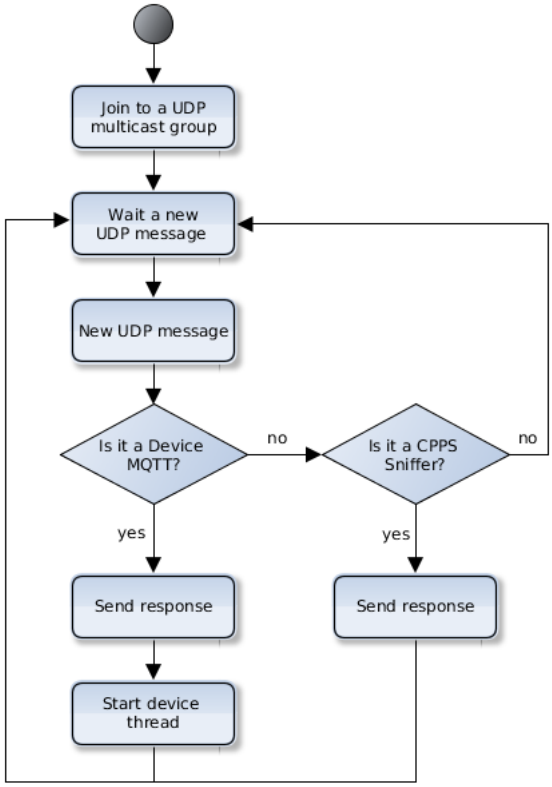

Figura 3.11: Diagrama de atividade da deteção de novos *CPPS Sniffers*/dispositivos.

espera de receber um nova mensagem, sendo que quando esta chega é verificado se se trata de um *CPPS Sniffer* ou dispositivo. Caso se trate de um *CPPS Sniffer* é enviada a respetiva resposta para este. Caso se trate de um dispositivo é enviada a resposta e é iniciada uma *thread* responsável pela gestão desse dispositivo. Após o processamento deste pedido, a *thread* volta ao estado em que fica à espera de receber novas mensagens.

Para o bom funcionamento deste mecanismo é necessário o envio de um pacote UDP da parte da nova componente. No caso do *CPPS Sniffer* o envio deste pacote já está incorporado no programa. Quanto aos dispositivos, caso se pretenda utilizar este mecanismo, é necessário implementar uma função responsável por enviar este pacote.

3.12 Mecanismos de deteção de falhas

Num sistema distribuído é crucial utilizar um sistema de deteção de falhas, pois é necessário saber quais as componentes da rede que estão a funcionar incorretamente, e reconfigurar a rede consoante essas falhas. Deste modo, foi necessário implementar no *CPPS Sniffer* certos mecanismos que permitissem verificar o estado dos restantes.

3.12.1 *Ping* entre *CPPS Sniffers*

Um dos métodos utilizados consiste na implementação de um sistema de troca de mensagens (*pings*) entre *CPPS Sniffers*, que permite inferir se o emissor da mensagem está a funcionar. Este processo baseia-se na implementação de um temporizador periódico, que dentro de uma certa periodicidade, verifica de quais *CPPS Sniffers* recebeu uma mensagem de *ping*. Caso não tenha recebido nenhuma mensagem proveniente de um certo *CPPS Sniffer* este remove-o da lista e envia um pedido de remoção para o resto da rede.

Os destinatários das mensagens de *ping* diferem caso se trate de um *CPPS Sniffer* remoto ou local. No caso de se tratar de um *CPPS Sniffer* local, este apenas tem de enviar *pings* para os *CPPS Sniffers* da sua rede local, enquanto o *CPPS Sniffer* remoto tem de enviar *pings* para os *CPPS Sniffers* da mesma rede local e para todos os *CPPS Sniffers* remotos existentes.

3.12.2 Perda de conexão com o *broker*

Outro dos métodos utilizados é baseado na perda de conexão com o *MQTT Broker* ao qual está ligado. Este método é executado sempre que é perdida a conexão com o *MQTT Broker*. Inicialmente tenta conectar-se outra vez ao *MQTT Broker*, sendo que caso não seja possível, é enviado um pedido de remoção do *CPPS Sniffer*/dispositivo para o resto da rede. Este processo é utilizado também quando não se consegue conectar inicialmente a um *MQTT Broker*, seja ao enviar uma mensagem de *ping* ou outra informação qualquer.

3.13 Resumo do capitulo

Para concluir, destaca-se que esta solução é robusta o suficiente para cumprir os requisitos que lhe foram impostos, no âmbito da gestão de redes de dispositivos IoT. Assim esta solução é capaz de comunicar via MQTT com o resto de componentes presentes na rede. Aquando do desenvolvimento desta solução, teve-se em conta que esta apresentasse características cruciais, como a facilidade de utilização por parte de utilizador, elevada robustez em situações de sobrecarga da rede e independência da plataforma onde executa o projeto.

Assim, este capitulo tem por objetivo documentar o projeto desenvolvido com o auxilio dos mais diversos diagramas UML (classes, sequência, atividades ou pacotes). A partir destes diagramas é possível inferir como se comporta o *CPPS Sniffer* quando sujeito a diferentes condições externas, e como foi estruturado o projeto.

Capítulo 4

Testes e Resultados

O processo de teste é uma fase crucial no desenvolvimento de *software*, através destes testes é possível corrigir erros ao nível do código ou da arquitetura. No entanto, os testes não existem apenas para detetar falhas ao nível do *software*, estes permitem também certificar o projeto desenvolvido, além de facilitar a otimização de todo o projeto. Assim neste capitulo vão ser apresentados os testes efetuados ao *CPPPS Sniffer*, por forma a validar a plataforma desenvolvida nas diferentes arquiteturas utilizadas (local ou remota), bem como em situações de fadiga (número elevado de dispositivos) de modo a perceber-se como se comporta o *CPPS Sniffer*. Além destes testes foram efetuados também testes de modo a perceber a qualidade do código desenvolvido no âmbito deste projeto.

Além da apresentação dos resultados obtidos este capítulo foca-se também na interpretação e comparação destes resultados. Desta forma estes resultados permitem inferir quais as vantagens e desvantagens da utilização do *CPPS Sniffer* em detrimento das outras plataformas ou arquiteturas IoT. Bem como, em que situações/requisitos se deve utilizar uma plataforma IoT ou o *CPPS Sniffer*.

4.1 Análise do código

De forma a obter métricas sobre o desenvolvimento deste projeto de software, foi utilizada a plataforma *codacy* [3], que permite através do repositório hospedado no *github*, adquirir métricas como o índice de cobertura do código, a complexidade de cada classe, os trechos de código duplicado e os problemas encontrados ao nível da performance, da compatibilidade, da segurança e do estilo do código.

Ao efetuar esta análise foi possível melhorar substancialmente a robustez do código desenvolvido associado ao *CPPS Sniffer*. Assim inicialmente quando o projeto foi submetido, este continha 64 problemas, sendo que todos eles foram revertidos, de forma a otimizar o código. Quanto ao índice de cobertura do código, esta métrica permite inferir a percentagem de código testado, tendo o *CPPS Sniffer* apresentado 82% de cobertura do código, o que é um bom resultado, considerando que quando foi efetuado o teste não foram executados alguns métodos relativos à subscrição da

lista e à comunicação *muticast*, sendo que ambos os métodos foram testados posteriormente com sucesso. A causa destes 2 métodos não terem sido executados no teste de cobertura foi devido a este ter sido efetuado no *localhost*, com os *MQTT Brokers* dos diferentes *CPPS Sniffers* a serem simulados no *docker*, de forma a obter um endereço IP para cada *MQTT Broker*.

Quanto à complexidade do código, esta permite inferir a facilidade de compreensão de cada classe, o tipo de complexidade avaliada foi a complexidade ciclomática, que se baseia nos diferentes fluxos de execução existentes em cada classe. Desta forma, as classes com maior complexidade são apresentadas na Tabela - 4.1, onde também é representado o número de linhas de código e o índice de cobertura. Como se verifica, a classe com maior nível de complexidade é a *SnifferOperations* com o valor de 13. Apesar de ser um valor elevado, justifica-se pelo facto da classe implementar o protocolo responsável pela comunicação entre *CPPS Sniffers*. As restantes classes apresentam um nível mais baixo, com valores entre 1 e 6. No que se refere ao número de linhas de código, este projeto totalizou 3304 linhas, distribuídas por 27 classes.

Tabela 4.1: Classes com maior complexidade.

Classe	Cobertura	Complexidade	# Linhas
`/udp/CheckNetworkThread.java`	70%	5	103
`/list/sniffer/SnifferOperations.java`	69%	13	174
`/list/operations/TimeoutsChecker.java`	95%	6	108
`/list/operations/RegistDevice.java`	67%	6	134
`/udp/NetworkOperations.java`	82%	6	93
`/list/sniffer/SnifferThread.java`	91%	5	89
`/list/device/DeviceThread.java`	93%	6	74
`/InitSystem.java`	88%	6	141

4.2 Características das componentes utilizadas

Para simular uma rede minimamente complexa são necessários diferentes componentes físicos, desde *router's*, *switch* de rede e dispositivos ligados a essas redes. Assim, para efetuar os testes necessários para validar o *CPPS Sniffer*, foram utilizadas *Raspberry Pi* de dois diferentes modelos, descritos na Tabela - 4.2.

Tabela 4.2: Especificações de ambos os modelos de *Raspberry Pi* utilizados.

Parâmetro	Raspberry Pi Version 1	Raspberry Pi 2 Version 2
Arquitetura da CPU	armv6l	armv7l
Número de *cores* de processamento	1	4
Frequência de *clock*	700MHz	700MHz
Memória RAM	434MiB	923MiB
Interface de rede	*ethernet*	*ethernet*

Após uma breve análise da Tabela - 4.2, as conclusões retiradas são que a *Raspberry Pi Version 2* tem maior poder de processamento que a *Raspberry Pi Version 1*, pois apesar de utilizarem a mesma frequência de relógio, a *Raspberry Pi Version 2* tem 4 vezes mais o número de *cores* (4 core) do que a *Raspberry Pi Version 1* (1 core), bem como uma arquitetura mais recente. Desta forma, a *Raspberry Pi Version 2* tem a capacidade de efetuar mais operações do que a *Raspberry Pi Version 1* no mesmo intervalo de tempo. Relativamente à memória RAM instalada, a *Raspberry Pi Version 2* apresenta vantagens em relação ao modelo anterior, pois contém 923MiB contra 434MiB do modelo anterior. Assim, a *Raspberry Pi Version 2* dispõe de mais memória RAM, o que permite executar maior número de programas ou programas mais complexos (exigentes em termos de memória). Quanto a interfaces de rede ambos os modelos apenas suportam comunicação via *ethernet*, o que não constitui uma desvantagem, pois podem ser ligados diretamente ao *router* por cabo.

4.3 Testes gerais com *CPPS Sniffer* local

Inicialmente, em termos de testes gerais vai ser adotada uma arquitetura local simples, de forma a perceber conceitos básicos da comunicação entre *CPPS Sniffers* e dispositivos, bem como verificar se as funcionalidades básicas desta plataforma estão operacionais, como a comunicação M2M entre dispositivos e entre *CPPS Sniffers*, a publicação de dados pelos dispositivos ou a troca de mensagens entre *CPPS Sniffers*.

Assim para verificar a validade do *CPPS Sniffer* foram efetuados diferentes testes para perceber quais as suas qualidades e as suas limitações.

4.3.1 Arquitetura Utilizada

Inicialmente, a arquitetura adotada foi construida com base numa rede local. A utilização de redes locais, sem conexão à Internet, é muito comum nas fábricas atuais, pois é uma forma de assegurar que os dados não saem das suas instalações, nem são capturados por entidades mal intencionadas. Esta arquitetura é representada na Figura - 4.1.

Relativamente à estrutura em si existem dois *CPPS Sniffers* locais e dois dispositivos, cada um deles a ser executado numa *Raspberry Pi*. Para além disso, o *MQTT Device 1* utiliza um *MQTT Broker* próprio, enquanto que o *MQTT Device 2* é associado ao *MQTT Broker* do *MQTT Device 1*. Desta forma, testa-se a utilização de um *MQTT Broker* por parte de um dispositivo, bem como a comunicação M2M, entre os *MQTT Device 1* e *MQTT Device 2*.

4.3.2 Resultados

As métricas usadas para avaliar o *CPPS Sniffer* foram a carga por CPU, a utilização de memória *Heap* e a utilização de memória *Non Heap*. A carga de CPU consiste num parâmetro que transmite o esforço médio atual da CPU, sendo que os valores normais deste parâmetro variam entre 0 e 1. Quando estes valores são maiores que 1, o CPU está numa situação de sobrecarga,

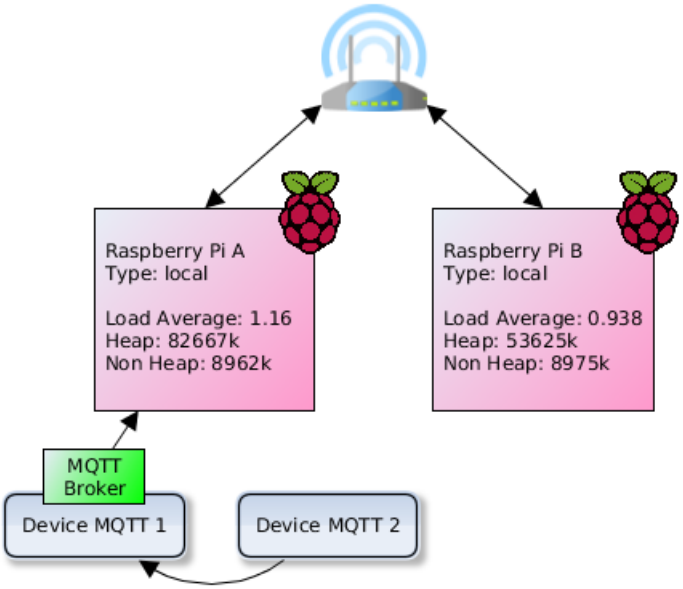

Figura 4.1: Arquitetura baseada *CPPS Sniffers* locais.

ou seja tem mais instruções para executar do que a capacidade deste. No caso, da máquina onde executa o código possuir mais do que uma CPU, este parâmetro continua válido, pois consiste na média dos valores medidos nos diferentes *cores* da máquina. Quanto à memória *Heap* e à *Non Heap*, estas permitem inferir sobre a utilização de memória RAM por parte do *CPPS Sniffer*. A principal diferença entre estes 2 tipos de memória é que a *Heap*, ao longo da execução do programa, vai sendo libertada pela máquina virtual. Por outro lado, a *Non Heap* é constituída pela *Stack* e pelas variáveis estáticas, sendo este tipo de memória estável até ao fim do processamento.

Para realizar a monitorização destas variáveis, usa-se um software que corre numa máquina independente, sendo considerado um dispositivo na rede como os outros, de forma a não interferir com a carga a ser executada na *Raspberry Pi* que executa o *CPPS Sniffer*. Contudo, não existe impedimento do dispositivo executar na mesma máquina em que está a executar o *CPPS Sniffer*.

Quanto aos testes efetuados, foram retiradas 2 amostras (teste 1 e 2) nas mesmas condições e no mesmo tipo de arquitetura constituída por dois *CPPS Sniffers* locais e dois dispositivos MQTT. Assim os valores da carga por CPU, da memória *Heap* e da memória *Non Heap* estão representados na Figura - 4.2.

Após a análise da Figura - 4.2 conclui-se que em termos de carga por CPU, o *CPPS Sniffer* com os dispositivos associados, apresenta valores mais elevados, aproximadamente 20%. Este valor é justificado com o facto do *CPPS Sniffer* subscrever os tópicos publicados no *MQTT Broker* associado ao dispositivo. Relativamente ao consumo de memória RAM num *CPPS Sniffer* local sem nenhum dispositivo associado ronda os 145MiB, enquanto que num *CPPS Sniffer* com dispositivos associados apresenta 175MiB, um acréscimo de 30MiB (20%). Este acréscimo deve-se ao aumento da memória *Heap*, pois a *Non Heap* mantém-se sempre constante (aproximadamente 90MiB), independentemente do número de dispositivos associados. Quanto à memória *Heap*, esta

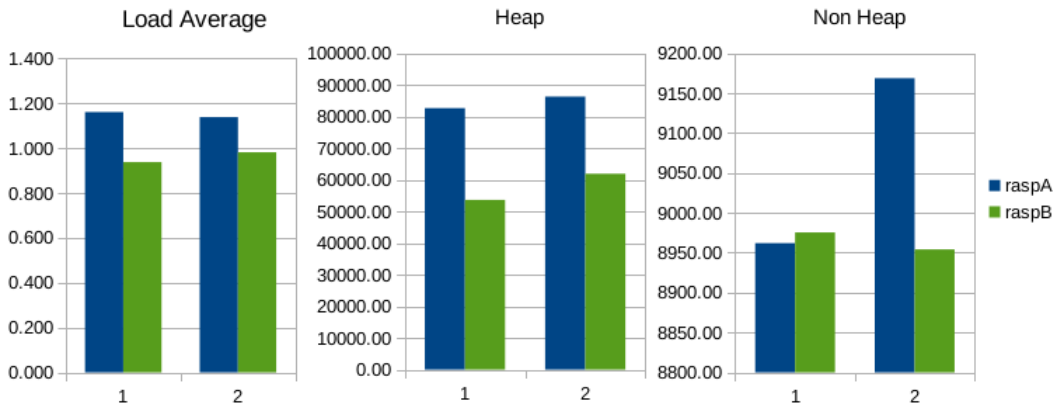

Figura 4.2: Testes efetuados na arquitetura local.

varia consoante o número de dispositivos adicionados. Neste caso, o *CPPS Sniffer* com dispositivos apresentava uma utilização de 85MiB, enquanto que o *CPPS Sniffer* sem dispositivos utilizava apenas 55MiB, ou seja quantos mais dispositivos adicionados maior é a utilização de memória *Heap*.

4.4 Testes gerais com *CPPS Sniffer* remoto

Depois de cumpridos os testes numa arquitetura simplificada (apenas local) é necessário validar o funcionamento do *CPPS Sniffer* numa arquitetura remota constituída por múltiplas redes locais. Para isso, em vez dos *CPPS Sniffers* estarem localizados em apenas um rede local, vão passar a existir múltiplas sub-redes e a comunicação entre estas vai passar a efetuar-se através da Internet.

Assim os testes, que antes foram efetuados numa arquitetura local, vão passar a ser efetuados numa arquitetura remota, com o recurso a *CPPS Sniffers* remotos, de forma a testar a comunicação entre múltiplas redes locais diferentes, através da Internet.

4.4.1 Arquitetura Utilizada

Assim para comprovar o comportamento do *CPPS Sniffer* remotamente, foi adotada a arquitetura apresentada na Figura - 4.3. Neste caso, existem quatro *Raspberry Pi*, 3 delas correspondem ao modelo antigo (*Raspberry Pi A, C e D*) e apenas uma corresponde ao modelo *Raspberry Pi Version 2* (*Raspberry Pi B*).

Em termos da distribuição das componentes existem duas redes locais reguladas, cada uma por um respetivo *router*. Assim na primeira rede local (*Network 1*) estão presentes as *Raspberry Pi* A e B, sendo que a *Raspberry Pi A* funciona como *CPPS Sniffer* remoto e a *Raspberry Pi B* como *CPPS Sniffer* local. O mesmo acontece na segunda rede local (*Network 2*), em que estão presentes as *Raspberry Pi* C e D, estando a *Raspberry Pi C* associada ao *CPPS Sniffer* remoto e a *Raspberry Pi D* ao *CPPS Sniffer* local.

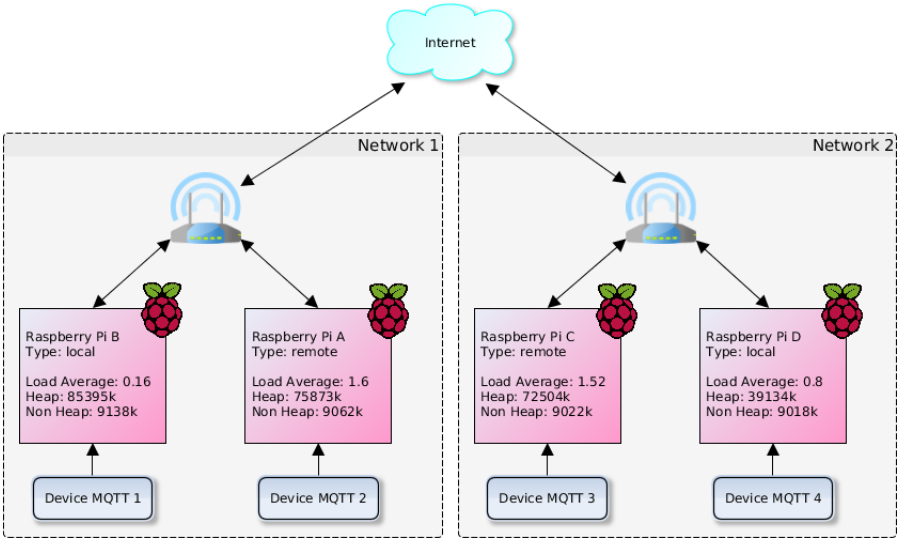

Figura 4.3: Arquitetura baseada *CPPS Sniffers* remotos.

Nesta situação foram também simulados dispositivos, cada um deles associado a um *CPPS Sniffer* e a executar na mesma (*Raspberry Pi*). Por outro lado também se considerou para testes a utilização da mesma arquitetura, só que apenas com *CPPS Sniffers*, sem dispositivos. Este caso serve como referência para estimar a quantidade de recursos que aumenta por cada dispositivo adicionado ao *CPPS Sniffer*.

4.4.2 *Brokers MQTT* remotos

De forma a permitir o funcionamento da rede IoT através da Internet, foram utilizados *MQTT Brokers* remotos, que são disponibilizados por um serviço *online*, semelhante aos serviços que disponibilizam bases de dados SQL. O site que disponibiliza os *MQTT Brokers* remotos é o CloudMQTT [2], que fornece vários planos, consoante as necessidades do utilizador (velocidade, número de ligações ou espaço disponível). Para este projeto optou-se pelo plano mais básico, que disponibiliza uma velocidade máxima de 10 Kbits/s e um limite máximo de 5 conexões. Como a utilização destes *MQTT Brokers* foi no intuito de realizar testes simples através da Internet, o plano mais básico cumpre os requisitos, sendo que para uma rede muito mais complexa ter-se-á de optar por um plano com maior número de conexões ou criar um servidor próprio.

Na Tabela - 4.3 estão representados os *MQTT Brokers* presentes na arquitetura descrita anteriormente. Neste caso existem dois *MQTT Brokers* remotos associados às *Raspberry Pi* A e C. Estes *MQTT Brokers* permitem testar uma situação real, pois estão hospedados em servidores remotos. Ambas as *Raspberry Pi* têm associado também um *MQTT Broker* local, como o resto das *Raspberry Pi* presentes na rede.

Tabela 4.3: Localização dos diferentes *MQTT Brokers* presentes na rede.

CPPS Sniffer	**Endereço IP/Host**	**Porta**	**Remoto**	**Localização**
Raspberry Pi A	m13.cloudmqtt.com	14795	✓	AWS Ireland
Raspberry Pi A	192.168.0.3	1883	✗	rede local 1
Raspberry Pi B	192.168.0.4	1883	✗	rede local 1
Raspberry Pi C	m21.cloudmqtt.com	17955	✓	AWS Northern Virginia
Raspberry Pi C	192.168.0.33	1883	✗	rede local 2
Raspberry Pi D	192.168.0.85	1883	✗	rede local 2

4.4.3 Resultados

De forma a obter os resultados pretendidos, foram efetuados diferentes testes na arquitetura remota composta por duas redes locais. Estes testes diferem entre eles, por forma a visualizar como o *CPPS Sniffer* se comporta em diferentes situações, assim os testes efetuados foram os seguintes:

- **Teste 1**: O primeiro teste foi efetuado com a arquitetura apresentada anteriormente, com a nuance de não conter dispositivos.
- **Teste 2**: Relativamente ao teste número 2, este foi efetuado nas mesmas condições que o teste 1, sendo que no teste 2 foi acrescentada a cada *Raspberry Pi* um dispositivo. Este dispositivo consiste num programa capaz de monitorizar certos parâmetros de uma *Raspberry Pi*.
- **Teste 3 e 5**: Os testes 3 e 5 foram efetuados sem dispositivos como no teste 1, divergindo apenas no facto de existir um processo a correr em segundo plano.
- **Teste 4**: Quanto ao teste 4, este foi efetuado nas mesmas condições que o teste 3 e 5, sendo que apenas foi acrescentado um dispositivo em cada *Raspberry Pi*.

Tal como no caso da arquitetura local, aqui foram utilizadas 3 métricas de avaliação, como forma de medir a performance da rede, nomeadamente a carga por CPU, a *Heap* e a *Non Heap*. A Figura - 4.4, representa os valores recolhidos carga por CPU, após efetuados os 5 testes descritos.

A partir dos dados recolhidos, fica demonstrado que a *Raspberry Pi Version 2* apresenta melhores resultados em comparação com a *Raspberry Pi Version 1*, pois apresenta menor carga por CPU (aproximadamente 10 vezes menor). Comparando apenas as *Raspberry Pi* do mesmo modelo, conclui-se que na maioria dos testes (2, 3, 4 e 5), as máquinas que executavam os *CPPS Sniffers* remotos (*Raspberry Pi A* e *Raspberry Pi C*) apresentam maior carga ao nível do processador (na ordem dos 25%, 40%) do que as máquinas que executam *CPPS Sniffers* locais. Isto justifica-se pela maior complexidade do *CPPS Sniffer* remoto. Quando é adicionado um dispositivo a cada componente da rede, ou seja comparando os testes 1 e 2 e os testes 4 e 5 constata-se que por cada dispositivo a carga por CPU aumenta entre 30% e 35%.

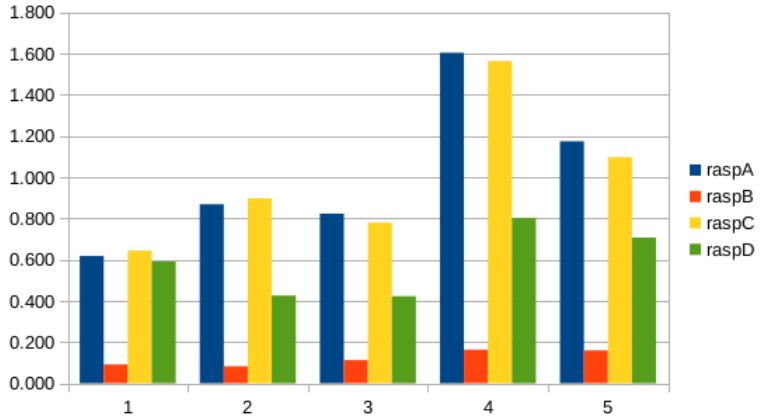

Figura 4.4: Carga da CPU em todas as *Raspberry Pi*.

Já quanto ao consumo de memória RAM, os dados relativos a este parâmetro são representados na Figura - 4.5.

Quando se somou a *Heap* e a *Non Heap* obteve-se o máximo de 180MiB, um valor relativamente normal para uma aplicação desenvolvida em Java, que pode tanto executar na *Raspberry Pi Version 1* (434MiB) como na *Raspberry Pi Version 2* (923MiB). A nível de memória *Non Heap* todos os *CPPS Sniffers* apresentam um valor semelhante, que ronda os 90MiB. Isto justifica-se com o facto deste tipo de memória ser persistente, ao contrário da *Heap*. Na memória *Heap*, através da comparação das *Raspberry Pi* C e D, conclui-se que o *CPPS Sniffer* remoto (*Raspberry Pi C*) utiliza aproximadamente mais 50%, 70% de memória *Heap*, que o *CPPS Sniffer* local (*Raspberry Pi D*). Quando são comparadas arquiteturas com dispositivos (testes 1 e 3) com arquiteturas sem dispositivos (testes 2 e 4) deduz-se que por cada dispositivo adicionado a utilização de *Heap* aumente entre 20% e 40%, enquanto que a *Non Heap* se mantém constante.

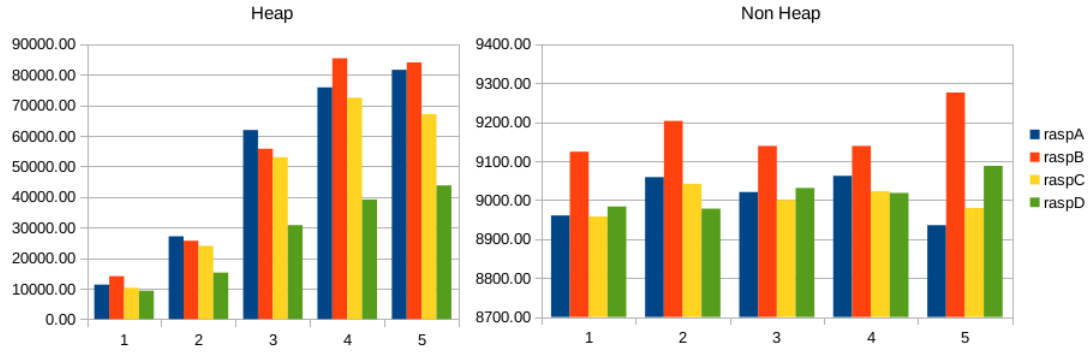

Figura 4.5: Utilização de Heap e resto da memória em todas as *Raspberry Pi*.

4.5 Testes quantitativos sobre o *CPPS Sniffer*

Os testes quantitativos constituem uma ótima métrica para avaliar o *CPPS Sniffer*. Através deste tipo de testes é possível quantificar e comparar numericamente os testes efetuados, de forma a retirar as respetivas conclusões. Os testes quantitativos efetuados vão permitir avaliar o tamanho das mensagens trocadas entre *CPPS Sniffers* e dispositivos, medir o atraso imposto pela rede nas mensagens, calcular o número máximo de dispositivos suportado por um *CPPS Sniffer* e avaliar o congestionamento da rede, em condições de fadiga extrema.

4.5.1 Tamanho das mensagens

O tamanho das mensagens é uma métrica de avaliação importante quando se compara protocolos de comunicação, pois quando se gera um grande volume de dados quanto maior for o tamanho dos pacotes maior será o congestionamento imposto à rede. Assim convém a utilização de um pacote que não introduza grandes *payloads*. Desta forma, nesta secção vão ser analisados a maior parte dos pacotes trocados entre *CPPS Sniffers* e dispositivos, em termos de tamanho dos pacotes enviados. Os dados obtidos estão apresentados na Tabela - 4.4, onde pode ser visualizado o tamanho de cada tipo de mensagem.

Tabela 4.4: Tamanho genérico das mensagens.

Tipo de mensagem	**Tamanho da mensagem**
Registo dispositivo	363 bytes
Registo *CPPS Sniffer*	316 bytes
Publicação dispositivo	136 bytes
Publicação lista	287 bytes
Conexão	139 bytes
Desconexão	68 bytes
Pedido para subscrever	80 bytes
Pedido para não subscrever	80 bytes
Ping entre *CPPS Sniffers*	150 bytes
Pedido de descoberta via UDP *multicast*	60 bytes
Resposta a pedido de descoberta via UDP *multicast*	45 bytes

A partir da análise da tabela é possível inferir quais os tipos de mensagem que têm maior tamanho. Dos tipos de mensagem utilizados alguns deles variam de tamanho consoante o número de componentes presentes na rede, sendo que um deles é o caso do tópico onde é publicada a lista de componentes. Pode-se verificar que este tipo de mensagem apresenta um tamanho na ordem dos 287 bytes, sendo que neste teste a rede apenas contém um *CPPS Sniffer*, ou seja este valor tende a aumentar. Quanto à mensagem de registo do dispositivo, o tamanho desta depende do número de tópicos que o dispositivo vai registar. No exemplo apresentado na tabela, o dispositivo regista 3 atributos, resultando disto um tamanho de 363 bytes. Relativamente ao resto de mensagens, o seu tamanho não varia substancialmente. Por exemplo, no caso do registo de um *CPPS Sniffer* o tamanho da mensagem depende do tipo de *CPPS Sniffer* (local ou remoto), sendo que no caso do

remoto a mensagem é ligeiramente maior. Importa referir que as mensagens UDP, utilizadas para a descoberta de componentes na rede, são de menor tamanho que as restantes, devido ao facto do UDP não utilizar um *header* muito extenso.

4.5.2 Atrasos impostos

De forma a testar o comportamento da rede, em termos de atrasos entre mensagens, foi efetuado um teste de maneira a verificar se o crescente número de dispositivos implicava um maior atraso nas mensagens. Para isso foi monitorizado um *CPPS Sniffer* local através da ferramenta *Wireshark* [8], sendo que os resultados desta monitorização estão representados na Figura - 4.6.

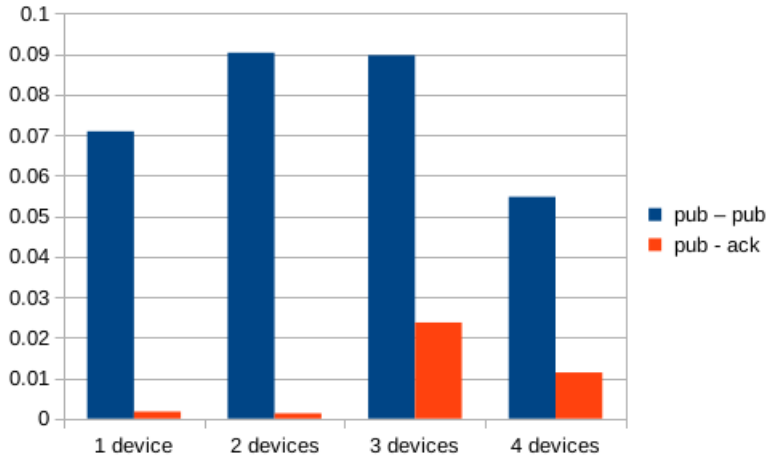

Figura 4.6: Atrasos medidos entre mensagens.

Da análise do gráfico conclui-se que o intervalo de tempo entre publicações consecutivas (*pub – pub*) mantém-se constante para este número de dispositivos, enquanto que o intervalo entre a publicação e a confirmação de receção (*pub – ack*) aumentou ligeiramente. Importa referir que o atraso imposto às mensagens pela rede é mais próximo da realidade no caso da troca de mensagens *pub – ack*, pois mal é recebida a publicação é imediatamente enviada a respetiva confirmação, ao contrário do que acontece entre as mensagens de publicação (*pub – pub*), pois entre 2 publicações existe um curto processamento por parte da máquina que as está a efetuar. Os dispositivos utilizados neste teste utilizam todos eles um *MQTT Broker* próprio, pelo que serviu também para testar esta funcionalidade.

4.5.3 Número máximo de dispositivos

De forma a medir o número máximo de dispositivos suportados por um *MQTT Broker*, foi projetado um teste capaz de simular esse processo. Este teste consiste em executar um *CPPS Sniffer* local numa *Raspberry Pi Version 1*, enquanto que noutra *Raspberry Pi* estão de 10 em 10 segundos a ser criados novos dispositivos que se vão ligar ao *CPPS Sniffer* local. Durante este processo foi monitorizada a carga por CPU a que o *CPPS Sniffer* estava sujeito, estando os resultados desta métrica representados na Figura - 4.7.

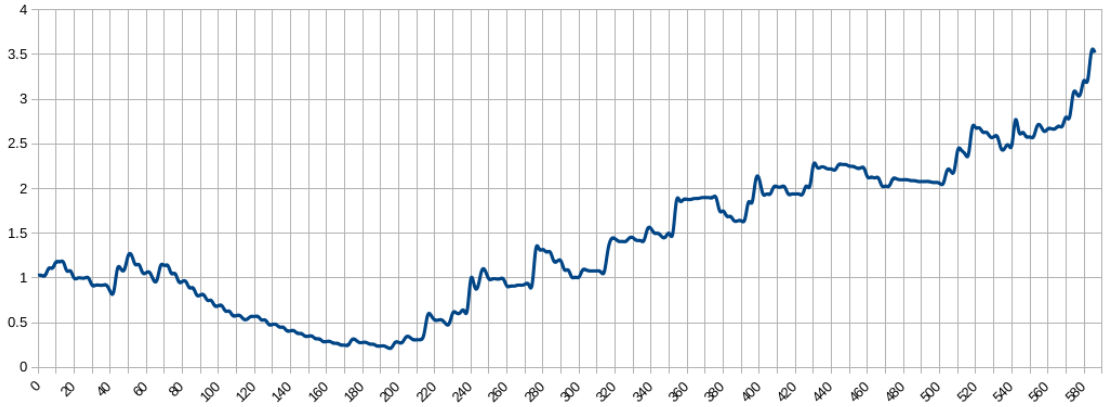

Figura 4.7: Evolução da carga por CPU, quando são constantemente adicionados dispositivos, a uma *Raspberry Pi Version 1*.

Analisando a evolução da carga por CPU do *CPPS Sniffer*, durante um período de 590 segundos, os primeiros 90 segundos descrevem o tempo que levou o *CPPS Sniffer* a carregar e a esperar que chegasse a primeira conexão de um dispositivo. Quanto aos dispositivos usados neste teste, foram simulados no total 50 dispositivos, que foram instanciados de 10 em 10 segundos o que permite inferir que a partir dos 90 segundos, a cada 10 segundos era adicionado um dispositivo, até ao teste terminar aos 590 segundos.

Em termos de comportamento do *CPPS Sniffer*, durante os primeiros 90 segundos, apresenta um comportamento estável a rondar o valor de 1 como carga por CPU. A partir dos 90 segundos, desce suavemente até aos 200 segundos (11 dispositivos), onde a carga por CPU começa a subir de forma linear a uma razão de 0.08 por dispositivo adicionado, ou seja a partir do dispositivo 11 por cada dispositivo adicionado a carga por CPU aumenta 0.08. Ao terminar o teste o *CPPS Sniffer* apresentava valores muito elevados de carga por CPU (aproximadamente 3.5), sendo que o mais recomendado é estar abaixo de 1. Desta forma, o ideal é ter no máximo 15 a 17 dispositivos associados a um *CPPS Sniffer*, sendo que o número ideal é de 11 dispositivos. De salientar que neste caso, cada dispositivo publicava dados de 3 atributos, multiplicando isto por 11 dispositivos resulta em 33 atributos, ou seja estes 11 dispositivos iniciais equivalem a um dispositivo com 33 atributos. Por isso, quando é utilizado o *CPPS Sniffer* tem de se ter em conta que não só o número de dispositivos é importante, mas também o número de atributos que estes utilizam.

4.5.4 Congestionamento da rede

Uma das formas de avaliar as limitações da solução desenvolvida é testar um grande volume de dados manipulados. De forma a simular este processo foi utilizado um teste descrito na subsecção 4.5.3, em que através do uso da ferramenta *Wireshark* [8] foram capturados todos os pacotes enviados e recebidos pela máquina onde que executava o *CPPS Sniffer* local, enquanto noutra máquina são criados novos dispositivos de 10 em 10 segundos, que são automaticamente associados

ao *CPPS Sniffer* já existente. Assim os resultados deste teste estão representados na Figura - 4.8, que apresenta o número de pacotes enviado ou recebido por segundo.

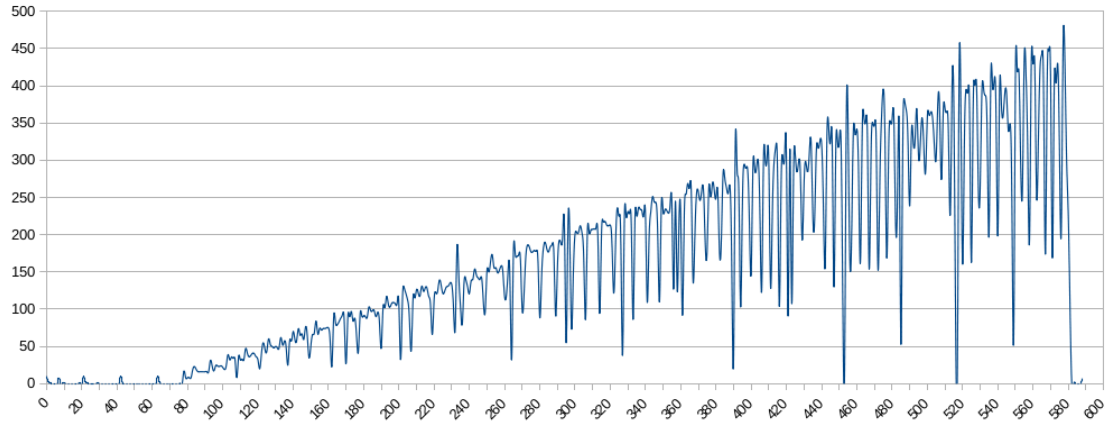

Figura 4.8: Congestionamento da rede ao longo de 600 segundos, medido em pacotes por segundo.

A partir da análise do gráfico conclui-se que é a partir dos 80 segundos que começa a execução do programa que simula os múltiplos dispositivos. Desde esse momento, a taxa de pacotes recebidos/enviados por segundo aumenta de forma proporcional, o que faz sentido, pois os dispositivos também são criados de uma forma proporcional (de 10 em 10 segundos acrescenta-se um dispositivo). Relativamente, à proporção a que o gráfico aumenta, por cada dispositivo acrescentado o número de pacotes por segundo aumenta em 9 (declive de 0.92). Estes valores não estão muito distantes da teoria, pois cada dispositivo publica dados de 3 tópicos, a cada segundo, sendo que para efetuar cada publicação são necessários 2 mensagens (*publish* e *acknowledge*). Desta forma, teoricamente são geradas 6 mensagens, contando com retransmissão de pacotes perdidos e outras possíveis falhas no envio de pacotes, o valor obtido de forma prática está bastante próximo do valor teórico.

Quanto aos primeiros 80 segundos, esse foi o tempo que levou a iniciar a *CPPS Sniffer*, estando este valor enquadrado nos descritos anteriormente. Este teste permitiu saber como se comportava a rede ao crescente número de dispositivos, sendo que a partir dos resultados, é deduzido que por cada dispositivo adicionado, o valor dos pacotes por segundo, aumenta em 9 pacotes.

4.6 Discussão dos resultados

Após serem apresentados todos os resultados dos testes efetuados no *CPPS Sniffer* é a altura de discutir e comparar esses resultados com os previamente discutidos na revisão bibliográfica, que utilizaram diferentes protocolos, ou até mesmo diferentes plataformas IoT. Para além disso convém também discutir as vantagens e desvantagens da utilização de uma arquitetura centralizada ou distribuída.

4.6.1 Comparação com plataformas IoT

Após serem apresentados todos os resultados dos testes efetuados no *CPPS Sniffer* é a altura de discutir e comparar esses resultados, com resultados obtidos em outras implementações, que utilizaram diferentes protocolos. Para além disso convém também discutir as vantagens e desvantagens da utilização de uma arquitetura centralizada ou distribuída.

De forma a comparar as plataformas IoT descritas na revisão bibliográfica com o *CPPS Sniffer* é necessário estabelecer quais as características que devem ser avaliadas. Numa primeira fase deve ser abordada o tipo de arquitetura adaptado tanto no caso das plataformas IoT como no caso do *CPPS Sniffer*, posteriormente devem ser considerados os protocolos suportados por ambos, bem como a facilidade de integração de novas plataformas, deve também ser considerada a utilização de bases de dados pela plataforma, finalmente devem ser comparados os resultados obtidos nos testes com o *CPPS Sniffer*, com os relacionados com as outras plataformas IoT.

Em termos de arquitetura, a maioria das plataformas IoT documentadas na revisão bibliográfica adotava uma arquitetura centralizada com a exceção de alguns deles que possuíam ou estavam a tentar desenvolver soluções para implementar ao nível da IoT *gateway*, como o *Linksmart* ou o *FIWARE* de maneira a descentralizar a sua arquitetura. Como o principal objetivo desta dissertação passava por desenvolver uma aplicação completamente distribuída e a maioria destas plataformas não cumpria este requisito na plenitude, sendo que este fator constituiu uma grande desvantagem na escolha ou não de uma plataforma IoT das referenciadas para a solução final.

O número de protocolos suportados por uma plataforma IoT é crucial para avaliar a flexibilidade desta plataforma. Neste particular, o *CPPS Sniffer* sai a perder na comparação com algumas plataformas, pois o *CPPS Sniffer* apenas suporta o protocolo MQTT, enquanto que as outras plataformas chegam a suportar 3 protocolos diferentes (FIWARE, *Microsoft Azure* ou *SiteWhere*). Um número maior de protocolos suportados por uma plataforma aumenta os níveis de compatibilidade com um maior número de dispositivos, o que constitui uma vantagem.

A integração de plataformas externas normalmente efetuada através de uma API, que por norma utiliza o protocolo REST. Várias plataformas IoT utilizam esta API, para integrar outras plataformas, exemplos disso são o FIWARE, o *SiteWhere* ou o *Linksmart*, no caso do *CPPS Sniffer*, como apenas é suportado o protocolo MQTT, apenas é possível integrar plataformas que comuniquem através deste protocolo.

A utilização de uma base de dados, seja relacional ou não-relacional é crucial para o armazenamento tanto dos registos dos dispositivos e das plataformas externas, como também dos dados produzidos por estes. No caso do *CPPS Sniffer* é utilizada uma base de dados distribuída (cada *CPPS Sniffer* possui a sua própria versão), constantemente atualizada entre todos *CPPS Sniffers*. Este tipo de base de dados constitui uma grande diferença quando comparada com as utilizadas pelas restantes plataformas IoT, que por norma são centralizadas. A utilização deste tipo de base de dados constitui uma vantagem na medida em que acrescenta maior redundância à plataforma e maior tolerância a possíveis falhas. No entanto possui desvantagens, pois de momento a base de dados distribuída apenas tem a capacidade de armazenar os registos das componentes presentes

na rede, não sendo capaz de guardar os dados produzidos pelos dispositivos, o que constitui uma desvantagem quando comparada com as outras plataformas IoT.

Relativamente à comparação entre os resultados obtidos no *CPPS Sniffer* e os obtidos em outras arquiteturas e com outras plataformas IoT, a comparação torna-se complicada pois as condições de teste são sempre diferentes (componentes diferentes).

4.6.2 Comparação com outras arquiteturas

Quando comparadas as arquiteturas centralizadas e distribuídas surgem certas vantagens e desvantagens da utilização de uma ou de outra. Assim nos próximos tópicos vão ser apresentadas as vantagens da utilização de um sistema distribuído em detrimento dos sistemas centralizados.

- **Preço**: Em termos de preço das componentes utilizadas estas tendem a ser mais baratas em sistemas distribuídos do que em sistemas centralizados, pois o processamento é repartido por todas as componentes presentes na rede, enquanto que em arquiteturas centralizadas é apenas encarregue uma componente pelo processamento de todos os dados produzidos. Ou seja, fica mais barato adquirir N equipamentos com menor poder de processamento do que um equipamento com elevado poder de processamento.

- **Confiabilidade**: A confiabilidade de um sistema permite inferir o grau de tolerância a falhas a que um sistema pode estar sujeito. Como é de esperar num sistema centralizado caso falhe uma das máquinas centrais da arquitetura o sistema não aguenta e colapsa. Enquanto que num sistema distribuído caso uma das máquinas falhe a rede reestrutura-se automaticamente de forma a continuar a funcionar. O cumprimento deste requisito é importante quando se pretende um sistema robusto.

- **Disponibilidade**: A disponibilidade é outra característica que está implícita na tolerância a falhas. Esta característica consiste em que quando uma máquina que disponibiliza um serviço é desconectada da rede existe a possibilidade de utilizar outra na mesma função.

- **Crescimento incremental**: Como foi referido anteriormente, a grande vantagem da utilização de um sistema distribuído é ser possível adicionar ou remover componentes da rede, conforme for a necessidade do utilizador, ou seja a capacidade de realocar recursos conforme as necessidades.

Por outro lado as desvantagens da utilização de um sistema distribuído, vão desde a segurança, à falta de *software*. Desta forma, nos tópicos seguintes vão ser descritos as desvantagens, com que os programadores tentam superar quando desenvolvem sistemas distribuídos.

- ***Software***: Como foi descrito anteriormente é difícil encontrar *software* qualificado nesta área, pois é relativamente recente além de ser mais complexo desenvolver *software* distribuído do que centralizado.

- **Congestionamento da rede**: Outra limitação encontrada é o congestionamento causado à rede, pois no normal funcionamento dos sistemas distribuídos é necessária a constante troca de dados entre as máquinas pertences à rede, pelo que o custo em termos de congestionamento da rede é maior em sistemas distribuídos do que em sistemas centralizados.
- **Segurança**: Em termos de segurança, os sistemas distribuídos estão mais expostos a fugas de dados, pois existe mais tráfego entre as suas componentes do que num sistema centralizado. Contudo, nem os sistemas distribuídos, nem os centralizados estão livres de sofrer fugas de informação, dependendo isto muito do projeto.

A partir das vantagens e desvantagens apresentadas acima, é possível concluir que os sistemas distribuídos são uma tecnologia com grande potencial de crescimento nos próximos anos, pois ainda não existe *software* suficientemente maduro neste sector e vantagens que resultam da utilização deste tipo de *software* são muito aliciantes.

Em termos de comparação entre os sistemas centralizados e sistemas distribuídos é impossível dizer que um é melhor que o outro, pois cada um deles tem vantagens e desvantagens. Assim a escolha entre um sistema distribuído ou centralizado depende das características pretendidas para o sistema, por exemplo os sistemas centralizados são mais fáceis de desenvolver em termos de *software*, no entanto o *hardware* utilizado nestes é mais caro do que o utilizado em sistemas distribuídos.

4.6.2.1 Comparação com o IDS

O *Industrial Data Space* (IDS) constitui uma arquitetura IoT muito completa e robusta, que tem como objetivo unir todos os sectores de uma determinada área, por forma a otimizar todos os processos. Desta forma, a aplicação desenvolvida no âmbito desta dissertação cobre algumas das funcionalidades desenvolvidas no projeto do IDS.

A descoberta de dispositivos/serviços no IDS é feita através de um *broker* centralizado, que funciona como uma base de dados, onde todos os dispositivos/serviços efetuam os seus registos, publicam os seus dados e através de *queries* podem efetuar a dita descoberta de serviços/dispositivos. Já no projeto do *CPPS Sniffer* esta funcionalidade foi desenvolvida de forma distinta, em vez de centralizada, é distribuída, ou seja em qualquer nó da rede onde exista um *CPPS Sniffer* o dispositivo/serviço tem acesso a todo o resto de componentes presentes na rede, bem como as suas localizações.

Quando é analisada a arquitetura do IDS a componente com mais semelhanças, para com o *CPPS Sniffer* é sem dúvida o conector local ou via Internet. Esta componente faz a conexão entre os dispositivos presentes no chão de fábrica e as outras componentes, podendo estas ser plataformas externas, bases de dados, etc. A grande diferença entre esta componente e o *CPPS Sniffer* é que o *CPPS Sniffer* para além de fazer a conexão entre os dispositivos e as outras plataformas faz também a gestão dos registos dos dispositivos e das plataformas externas, ou seja disponibiliza funcionalidades de gestão e descoberta de dispositivos ou serviços. No fundo, o *CPPS Sniffer* engloba as funcionalidades do *data broker* e do conector local ou remoto presentes na arquitetura

do IDS numa única plataforma distribuída. Enquanto que uma das desvantagens do *CPPS Sniffer* quando comparado ao IDS consiste no facto do *CPPS Sniffer* não integrar uma base de dados de forma a guardar os dados produzidos pelos dispositivos, sendo isto uma desvantagem na medida em que o armazenamento dos dados neste tipo de plataformas é crucial na obtenção de resultados para otimizar os processos.

Relativamente, às restantes componentes utilizados na arquitetura do IDS, os dispositivos presentes no chão de fábrica estão associados a uma rede local, tal como acontece na arquitetura utilizada pelo *CPPS Sniffer*, enquanto que as plataformas integradas no IDS funcionam com serviços/aplicações presentes na *AppStore*, contrariamente à arquitetura do *CPPS Sniffer*, onde apenas é possível integrar plataformas externas que comuniquem através do protocolo MQTT. Relativamente, aos protocolos utilizados a arquitetura do IDS não especifica quais os tipos de protocolos suportados.

No que se refere à reestruturação automática da rede e à tolerância a falhas, a arquitetura utilizada no *CPPS Sniffer* utiliza mecanismos de reestruturação da rede, caso falhe algum dos *CPPS Sniffers*, enquanto que no caso do IDS não são mencionados nenhuns mecanismos que promovam redundância ao sistema em caso de falha de alguma das componentes. A fácil reestruturação da rede utilizada pelo *CPPS Sniffer* aumenta também a facilidade de aumentar a rede (acrescentar dispositivos ou plataformas externas).

4.7 Resumo do capitulo

Neste capitulo foi possível verificar que o *CPPS Sniffer* cumpriu os requisitos estabelecidos ao passar nos testes efetuados tanto numa arquitetura local, como uma arquitetura remota com múltiplas redes locais. Para além de cumprir os testes gerais, o *CPPS Sniffer* efetuou também uma série de testes quantitativos, que permitiram quantificar o tamanho dos pacotes trocados entre as diferentes componentes da rede, o atraso imposto às mensagens, o número máximo de dispositivos suportados pelo *CPPS Sniffer* e o congestionamento da rede. Estes testes foram cruciais, na medida em que permitiram inferir como se comportava o *CPPS Sniffer*, quando submetido a diferentes situações de stress.

Posteriormente, foram analisados os resultados obtidos, de forma a serem comparados com os resultados descritos na revisão bibliográfica. Na comparação com as plataformas IoT descritas conclui-se que o *CPPS Sniffer*, constitui um produto inovador neste mercado, pois a maioria das plataformas descritas adota uma arquitetura centralizada, enquanto que o *CPPS Sniffer* é uma plataforma distribuída. No entanto, não foram só apresentadas vantagens em relação ao *CPPS Sniffer*, pois ainda é uma plataforma recente, que ainda não suporta o armazenamento da dados provenientes dos dispositivos, como também não tem uma API para integração de outras plataformas que não utilizem MQTT como protocolo (por exemplo, plataformas que comuniquem através de REST).

Desta forma, torna-se importante efetuar a comparação entre as arquiteturas centralizada e distribuída. Esta comparação apresenta tanto as vantagens como as desvantagens de ambas as

arquitetura, concluindo-se que a escolha entre uma arquitetura centralizada ou distribuída, baseia-se principalmente em quais são os requisitos da plataforma, não sendo um tipo de arquitetura melhor ou pior que o outro. Em termos de arquitetura é também efetuada uma comparação com a arquitetura utilizada no IDS, resultando desta comparação semelhanças e diferenças entre as arquiteturas, bem como vantagens e desvantagens.

Capítulo 5

Conclusões e Trabalho Futuro

Após descrever os testes efetuados e discutir os resultados obtidos é a altura de apresentar as últimas conclusões do trabalho efetuado e discutir o trabalho que se pretende desenvolver no futuro. A partir das conclusões retiradas infere-se quais dos requisitos estabelecidos se cumpriram e quais as contribuições do *CPPS Sniffer* para esta área.

Outro ponto muito importante a analisar é o trabalho futuro, não só para descrever o que faltou fazer nesta dissertação, mas também para apresentar novas ideias, que futuramente podem melhorar a solução encontrada. Assim na descrição do trabalho futuro são abordas algumas das funcionalidades que acrescentariam valor ao *CPPS Sniffer*.

5.1 Contribuições

O *CPPS Sniffer* foi desenvolvido com o objetivo de ser uma plataforma completamente distribuída, pois comparando com as outras plataformas, a maioria destas utiliza uma arquitetura centralizada, constituindo esta plataforma uma grande inovação. Deste ponto de vista, a plataforma distribuída, *CPPS Sniffer*, cumpre os requisitos necessários, que qualquer aplicação deste tipo deve cumprir, constituindo estes tópicos algumas das vantagens que o *CPPS Sniffer* tem em relação às plataformas IoT mais centralizadas.

- **Transparência:** A transparência consiste na visibilidade de todas as componentes presentes na rede global, independente da sub-rede a que pertencerem. Assim, este requisito foi cumprido, pois através dos testes efetuados, ficou saliente que os dispositivos conseguiam ver-se entre eles mesmo através da Internet estando os dispositivos em diferentes rede locais.

- **Flexibilidade:** A flexibilidade consiste na facilidade de integração de novas componentes, podendo estas ser simples dispositivos ou outras plataformas mais complexas. Para isso foi desenvolvida a descoberta automática de dispositivos através de IPs *multicast*, facilitando assim a auto-configuração dos dispositivos na rede, o que aumenta a flexibilidade da plataforma. Relativamente, aos protocolos suportados apenas é utilizado o protocolo MQTT

constituindo isso uma limitação em termos de flexibilidade para com os dispositivos que utilizem outros protocolos.

- **Desempenho:** Relativamente, ao desempenho da plataforma foi possível executar o *CPPS Sniffer* em dispositivos com baixo poder de processamento (*Raspberry Pi Version 1*). Em termos de desempenho da rede foram adicionados constantemente novos dispositivos sem que a rede ficasse congestionada.

- **Tolerância a falhas:** O *CPPS Sniffer* é tolerante a falhas pois quando ocorre uma falha num dos *CPPS Sniffers* existentes, a rede responde com a automática reestruturação. Um dos contratempos neste particular foi o facto deste tipo de mecanismos não ter sido aplicado a dispositivos, facilitando assim a deteção de dispositivos com falhas.

- **Escalabilidade:** Em termos de escalabilidade, a plataforma permite a constante adição de nos *CPPS Sniffers* ou dispositivos à rede global, sem que isso interfira com o bom desempenho da rede.

Em suma, conclui-se que o *CPPS Sniffer* abarca todas as características, que uma plataforma distribuída, deve apresentar, transparência, flexibilidade, desempenho e tolerância a falhas. Em termos dos requisitos estabelecidos, estes foram cumpridos, pois verificou-se no desempenho do *CPPS Sniffer*, que este é capaz de operar tanto numa simples rede local, como em múltiplas redes locais através da Internet. Para além de confirmados estes testes, também foram validadas funcionalidades que permitiam a gestão de dispositivos entre as diferentes redes e a descoberta de dispositivos e serviços tanto a nível local como a nível remoto. Assim conclui-se, que a hipótese descrita anteriormente ficou confirmada, pois o *CPPS Sniffer* apresenta um bom desempenho tanto a nível de cooperação com outros *CPPS Sniffers*, como também com dispositivos ou plataformas externas, seja em arquiteturas locais ou remotas.

Quando comparado com outras plataformas IoT existentes no mercado, o *CPPS Sniffer* tem características únicas diferentes de todas as outras plataformas. Estas características facilitam a integração seja de redes locais, dispositivos ou plataformas externas, de forma a que o trabalho de configuração de qualquer uma destas componentes fique mais facilitado. De todas estas características nos tópicos abaixo são descritas as mais importantes.

- **Facilidade de integração de novas componentes:** A integração de novas componentes na rede, está tão facilitada que para uma nova componente que se queira juntar à rede, da forma que esta não precisa de saber nem a estrutura da rede, nem as localizações do *CPPS Sniffers*. Por exemplo, caso um técnico pretenda integrar um simples programa (por exemplo, análise de dados), este apenas precisa de instalar um *CPPS Sniffer* no seu computador e a partir desse momento o *CPPS Sniffer* liga-se automaticamente à rede existente, bastando ao técnico publicar/subscrever mensagens no *localhost*, que o *CPPS Sniffer* fica encarregue da difusão dos dados pela rede.

- **Mecanismos de cooperação entre os *CPPS Sniffers*:** O *CPPS Sniffer* ao ser uma plataforma completamente distribuída é possível que numa rede local existam vários, pelo que é necessário uma constante cooperação entre todos, de forma a manter a rede a funcionar corretamente, estes mecanismos são capazes de detetar falhas entre *CPPS Sniffers*, bem como partilhar informações cruciais entre eles (por exemplo, registo de novos dispositivos). Esta característica advém vantagens na medida, em que permite aumentar facilmente a rede em caso de esta estar congestionada em certo pontos.

- **Plataforma construída ao nível de *Fog Computing*:** Entre as plataformas IoT existentes, poucas funcionam ao nível de *Fog Computing*, sendo desta forma o *CPPS Sniffer* bastante inovador. O facto do *CPPS Sniffer* estar implementado a este nível, advém algumas vantagens interessantes à plataforma, entre elas o facto de estar mais fácil e rápido o acesso a dados por parte das componentes pertencentes à rede, além de facilitar a ampliação da rede como foi referido anteriormente.

5.2 Trabalho futuro

A nível de trabalho futuro as ideias apresentadas de seguida seguem uma filosofia de utilizar o *CPPS Sniffer* como base de uma arquitetura IoT, local ou remota, e a partir desta acrescentar funcionalidades estratégicas, que permitam facilitar a tarefa de acrescentar seja plataformas externas, ou outros tipos de dispositivos. Outras funcionalidades interessantes estariam relacionadas com o desenvolvimento de uma plataforma para monitorização das componentes da rede, com a utilização de bases de dados para armazenar o histórico de dados dos dispositivos ou com a melhora do processo de reestruturação da rede e do processo de descoberta de dispositivos.

- **Suporte para mais protocolos:** De forma, a aumentar o suporte a um maior número de dispositivos é crucial acrescentar suporte para mais protocolos como o CoAP, o AMQP ou o OPC-UA de maneira a aumentar a flexibilidade do *CPPS Sniffer*. Dos protocolos referidos, o que seria mais interessante de adicionar ao *CPPS Sniffer* é o OPC-UA, pois a nível de dispositivos industriais (por exemplo, PLC's) a sua utilização é praticamente um standard. Outro protocolo, interessante de integrar no *CPPS Sniffer* seria o CoAP, pois é um protocolo direcionado a dispositivos com poucos recursos computacionais.

- **API para integração de outras plataformas:** Na maioria das plataformas IoT descritas anteriormente, as plataformas externas são integradas através de uma API, que comunica através do protocolo REST, pelo que seria interessante devolver uma API deste tipo no âmbito do *CPPS Sniffer*, por forma a conceder maior flexibilidade à solução, na hora de integração de outras plataformas, pois neste momento o *CPPS Sniffer* apenas suporta a integração de plataformas que comuniquem via MQTT.

- **Melhorar processamento de dados local:** De momento, os dados provenientes dos dispositivos, sofrem um simples processamento (média dos valores) quando chegam ao *CPPS*

Sniffer. Este simples processamento pode ser melhorado, de forma a ter em conta os tipos de variáveis dos dados, ou até mesmo permitir guardar um histórico recente dos dados produzidos pelos dispositivos.

- **Integrar base de dados ao nível do *CPPS Sniffer*:** Como foi referido no ponto anterior, acrescentaria valor ao *CPPS Sniffer* a utilização de uma base de dados que armazenasse o histórico de dados dos dispositivos associados ao respetivo *CPPS Sniffer* e que posteriormente caso pretendido fosse possível aceder a esse histórico, através de uma API REST, ou até mesmo através de MQTT.

- **Desenvolver interface com o utilizador:** Para monitorização da estrutura da rede, é crucial utilizar uma plataforma deste tipo seja uma aplicação para *smartphone* seja uma página Web. A ideia da utilização de uma página Web seria mais útil, pois através de um navegador de computador presente na fábrica seria possível ver o estado atual da rede, o histórico dos dispositivos, entre outras variáveis. Para aceder à pagina das definições da rede e do *CPPS Sniffer* bastaria aceder ao domínio (endereço IP e porta) associado a um *CPPS Sniffer*. Além da monitorização da rede, esta interface gráfica seria importante para a gestão de todas as configurações associadas a *CPPS Sniffers*, como por exemplo realocar dispositivos entre *CPPS Sniffers* de forma a distribuir melhor a rede.

- **Melhorar processo de reestruturação da rede:** Neste momento já existe uma reestruturação relativa aos *CPPS Sniffers* quando um destes falha, no entanto os dispositivos associados a este *CPPS Sniffer* não são distribuídos pelos restantes, como também não são distribuídos de uma forma igualitária pelos *CPPS Sniffers* quando estes efetuam o primeiro registo. O que passaria a acontecer caso se implementasse esta funcionalidade seria que os dispositivos seriam distribuídos de uma forma inteligente e igualitária por todos os *CPPS Sniffers* e caso um *CPPS Sniffer* falhasse, os dispositivos associados a este seriam distribuídos pelos outros *CPPS Sniffers*.

- **Integrar plataforma IoT centralizada:** Caso fosse pretendido num dos nós seria possível adicionar uma plataforma IoT centralizada, das descritas anteriormente, de forma a adicionar redundância à plataforma e a permitir usufruir de todas as plataformas externas que algumas destas plataformas já têm integradas.

- **Identificar conferência para publicação dos resultados:** Por último, mas não menos importante, é identificar uma conferência para a publicação dos resultados. Assim, será necessário converter/resumir este documento, num artigo cientifico, que possa a vir ser publicado.

Anexo A

Manual de Instalação

O manual de instalação tem como função descrever todo o processo de instalação do *CPPS Sniffer*. Para efetuar este processo é necessário que a máquina onde vai executar o *CPPS Sniffer* tenha instalados o *Java Development Kit* (JDK) e o *Apache Ant*. O JDK é necessário tanto para a compilação do código Java como também para a sua execução, enquanto que o *Apache Ant* é apenas utilizado para compilar o projeto.

Após a instalação desta duas componentes é necessário descarregar o projetor hospedado no *github* [24]. Depois de efetuar a descarga do projeto o utilizador deverá editar o ficheiro JSON de configuração 3.5 de maneira a definir parâmetros importantes no funcionamento do *CPPS Sniffer*. Quando o *CPPS Sniffer* estiver configurado como pretendido o utilizador deverá abrir um terminal/linha de comandos na pasta base do projeto para o compilar e posteriormente executar o projeto. Para o processo de compilação é utilizado o *Apache Ant*, através do seguinte comando:

```
ant -Dnb.internal.action.name=rebuild clean jar
```

Caso seja apresentada no terminal/linha de comandos a mensagem "BUILD SUCCESSFUL", o utilizador poderá passar para o próximo passo que seria executar o projeto. Para executar o *CPPS Sniffer* é utilizado o seguinte comando:

```
java -jar dist/Sniffer.jar
```

Referências

[1] A. Chaudhary, S. K. Peddoju, and K. Kadarla. Study of internet-of-things messaging protocols used for exchanging data with external sources. In *Mobile Ad Hoc and Sensor Systems (MASS), 2017 IEEE 14th International Conference on*, pages 666–671. IEEE, 2017.

[2] CloudMQTT. Cloud mqtt, 2018. URL https://www.cloudmqtt.com.

[3] Codacy. Codacy, 2018. URL https://www.codacy.com.

[4] A. Corporation. Liksmart documentation, 2018. URL https://docs.linksmart.eu/.

[5] M. Fazio, A. Celesti, A. Glikson, and M. Villari. Exploiting the FIWARE Cloud Platform to Develop a Remote Patient Monitoring System. *Fifth International Workshop on Management of Cloud and Smart City Systems 2015 Exploiting*, pages 264–270, 2015.

[6] S. Forsstrom and U. Jennehag. A performance and cost evaluation of combining OPC-UA and Microsoft Azure IoT Hub into an industrial Internet-of-Things system. *GIoTS 2017 - Global Internet of Things Summit, Proceedings*, 2017. doi: 10.1109/GIOTS.2017.8016265.

[7] E. Foundation. Eclipse paho, 2016. URL https://www.eclipse.org/paho/clients/java/.

[8] W. Foundation. Wireshark go deep, 2018. URL https://www.wireshark.org/.

[9] J. Guth, U. Breitenbücher, M. Falkenthal, F. Leymann, and L. Reinfurt. Comparison of iot platform architectures: A field study based on a reference architecture. In *Cloudification of the Internet of Things (CIoT)*, pages 1–6. IEEE, 2016.

[10] P. Hu. A system architecture for software-defined industrial internet of things. In *Ubiquitous Wireless Broadband (ICUWB), 2015 IEEE International Conference on*, pages 1–5. IEEE, 2015.

[11] M. Iglesias-Urkia, A. Orive, M. Barcelo, A. Moran, J. Bilbao, and A. Urbieta. Towards a lightweight protocol for industry 4.0: An implementation based benchmark. In *Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM), 2017 IEEE International Workshop of*, pages 1–6. IEEE, 2017.

[12] S. Katsikeas, K. Fysarakis, A. Miaoudakis, A. Van Bemten, I. Askoxylakis, I. Papaefstathiou, and A. Plemenos. Lightweight & secure industrial IoT communications via the MQ telemetry transport protocol. *Proceedings - IEEE Symposium on Computers and Communications*, pages 1193–1200, 2017. ISSN 15301346. doi: 10.1109/ISCC.2017.8024687.

[13] B. Konieczek, M. Rethfeldt, F. Golatowski, and D. Timmermann. A distributed time server for the real-time extension of coap. In *Real-Time Distributed Computing (ISORC), 2016 IEEE 19th International Symposium on*, pages 84–91. IEEE, 2016.

[14] konker Labs. Konker documentation, 2018. URL http://developers.konkerlabs.com/.

[15] S. LLC. Sitewhere documentation, 2017. URL http://documentation.sitewhere.io/architecture.html.

[16] S. LLC. Sitewhere repository, 2018. URL https://github.com/sitewhere/sitewhere.

[17] J. A. López-Riquelme, N. Pavón-Pulido, H. Navarro-Hellín, F. Soto-Valles, and R. Torres-Sánchez. A software architecture based on FIWARE cloud for Precision Agriculture. *Agricultural Water Management*, 183:123–135, 2017. ISSN 18732283. doi: 10.1016/j.agwat.2016.10.020.

[18] P. Masek, J. Hosek, K. Zeman, M. Stusek, D. Kovac, P. Cika, J. Masek, S. Andreev, and F. Kröpfl. Implementation of True IoT Vision: Survey on Enabling Protocols and Hands-On Experience. *International Journal of Distributed Sensor Networks*, 2016, 2016. ISSN 15501477. doi: 10.1155/2016/8160282.

[19] T. Mizuya, M. Okuda, and T. Nagao. A case study of data acquisition from field devices using opc ua and mqtt. In *Society of Instrument and Control Engineers of Japan (SICE), 2017 56th Annual Conference of the*, pages 611–614. IEEE, 2017.

[20] N. Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. *2017 IEEE International Symposium on Systems Engineering, ISSE 2017 - Proceedings*, 2017. doi: 10.1109/SysEng.2017.8088251.

[21] OASIS. Amqp: Advanced message queuing protocol, 2015. URL https://www.amqp.org/.

[22] B. Otto, S. Auer, J. Cirullies, J. Jürjens, N. Menz, J. Schon, and S. Wenzel. Industrial Data Space. *Fraunhofer-Gesellschaft*, 2017, 2017.

[23] G. Peralta, M. Iglesias-Urkia, M. Barcelo, R. Gomez, A. Moran, and J. Bilbao. Fog computing based efficient iot scheme for the industry 4.0. In *Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM), 2017 IEEE International Workshop of*, pages 1–6. IEEE, 2017.

[24] E. Pereira. Sniffer, 2018. URL https://github.com/eliseu10/Fiware-Sniffer.

[25] J. Pfrommer and F. Palm. RESTful Industrial Communication With OPC UA. *IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS*, 12(5):1832–1841, 2016.

[26] A. Preventis, K. Stravoskoufos, S. Sotiriadis, and E. G. M. Petrakis. Personalized Motion Sensor Driven Gesture Recognition in the FIWARE Cloud Platform. *14th International Symposium on Parallel and Distributed Computing Personalized*, pages 19–26, 2015. doi: 10.1109/ISPDC.2015.10.

[27] A. Selva. Moquette mqtt broker, 2018. URL http://andsel.github.io/moquette/.

[28] W. Tärneberg, V. Chandrasekaran, and M. Humphrey. Experiences creating a framework for smart traffic control using AWS IOT. *Proceedings of the 9th International Conference on Utility and Cloud Computing - UCC '16*, pages 63–69, 2016. doi: 10.1145/2996890.2996911. URL http://dl.acm.org/citation.cfm?doid=2996890.2996911.

[29] C. Technology. Coap: Rfc 7252 constrained application protocol, 2014. URL http://coap.technology/.

[30] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan. Performance evaluation of mqtt and coap via a common middleware. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, pages 1–6. IEEE, 2014.

[31] I. D. Works. Mq telemetry transport (mqtt): V3.1 protocol specification, 2014. URL http://goo.gl/3tLZVj.